WORKSHOPS

ETAPS 2002

April 6–14, 2002
Grenoble, FRANCE

April 6–7

Designing Correct Circuits

(DCC'02)

Mary Sheeran (Chalmers University of Technology)
Tom Melham (University of Glasgow)

Organized by the Formal Methods Group of Chalmers University of Technology

UNIVERSITY *of* GLASGOW

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From – To)*
09-03-2004	Conference Proceedings	6 April 2002 - 14 April 2002

4. TITLE AND SUBTITLE

European Joint Conferences on Theory and Practice of Software (ETAPS)

5a. CONTRACT NUMBER
F61775-02-WF002

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

Conference Committee

5d. PROJECT NUMBER

5d. TASK NUMBER

5e. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Verimag UMR 5104 CNRS/INPG/UJF
Centre Equation
2, Avenue de Vignate
Gières F-38610
France

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

EOARD
PSC 802 BOX 14
FPO 09499-0014

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
CSP 02-5002

12. DISTRIBUTION/AVAILABILITY STATEMENT

Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES

April 6-7 – Designing Correct Circuits (DCC'02)

14. ABSTRACT

The Final Proceedings for European Joint Conferences on Theory and Practice of Software (ETAPS), 6 April 2002 - 14 April 2002

This is a computer and information science conference. Topics include compiler construction; programming language implementation; language design; specification, design, and analysis of programming languages and programming systems; fundamental approaches to software engineering; component-based software architectures; middleware systems for large scale heterogeneous software federation; formal modeling and specification techniques for component based software; mathematical models and methods for the specification, synthesis, verification, analysis, and transformation of sequential, concurrent, distributed, and mobile programs and software systems; tools and algorithms for the construction and analysis of systems.

15. SUBJECT TERMS

EOARD, software engineering, formal methods

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			PAUL LOSIEWICZ, Ph. D.
UNCLAS	UNCLAS	UNCLAS	UL	Many	19b. TELEPHONE NUMBER *(Include area code)* +44 20 7514 4474

02-5002

Preface

This volume contains material associated with the presentations at the Fourth International Workshop on Designing Correct Circuits, to be held on April 6^{th} and 7^{th}, 2002, in Grenoble, France. The workshop is a satellite event of the ETAPS group of conferences. Previous workshops in the informal DCC series were held in Oxford (1990), Lyngby (1992), and Båstad (1996). These meetings were all very stimulating events, and each made a contribution to building our research community.

The 2002 DCC workshop again brings together academic and industrial researchers in formal methods for hardware design and verification. It will allow participants to learn about the current state of the art in formally-based hardware verification and it is intended to spark debate about how more effective design and verification methods can be developed.

Much research in hardware verification now takes place in industry, rather than in academia. For the long term survival of our field, we must ensure that academics and industrial researchers continue to work together on the real problems facing microprocessor designers and those developing System on a Chip solutions. A major aim of the workshop is to open the necessary communication channels. With the speakers that DCC 2002 has attracted, it seems likely that the debate will be lively and productive! We look forward to two great days.

Acknowledgements

We would like to express our gratitude to the members of the programme committee for their work in selecting the presentations.

Mary Sheeran and Tom Melham
March 2002

Programme Committee

Per Bjesse (Prover Technology)
Koen Claessen (Chalmers)
John Hughes (Chalmers)
Matt Kaufmann (AMD)
Tim Leonard (Intel)
Tom Melham (Glasgow)
John O'Leary (Intel)
Mary Sheeran (Chalmers/Prover Technology)
Jean Vuillemin (ENS,Paris)
Dominique Borrione (TIMA/UJF,Grenoble)
Nicolas Halbwachs (IMAG,Grenoble)
Steve Johnson (Indiana)
John Launchbury (OGI)
Andy Martin (Motorola)
Alan Mycroft (Cambridge)
Gordon Pace (INRIA)
Satnam Singh (Xilinx)

Designing Correct Circuits 2002

6-7 April 2002, Grenoble

An ETAPS 2002 Satellite Workshop

Programme

Saturday, 6 April 2002

9:00-9:30 *Several talking points about hardware design and verification*
Carl Pixley, Ken Albin and John Havlicek (Synopsys and Motorola)

9:30-10:00 *Unifying Traditional and Formal Verification through Property Specification*
Harry Foster (Verplex)

10:00-10:30 *Verisym: A Tool for Array Verification*
Warren Hunt (IBM)

10:30-11:00 Coffee

11:00-11:30 *Lava: An Embedded Language for Structural Hardware Design*
Koen Claessen, Mary Sheeran, and Satnam Singh (Chalmers University of Technology and Xilinx)

11:30-12:00 *Generic Operators for Circuit Synthesis and Optimisation*
Jean Vuillemin (ENS, Paris)

12:00-14:00 Lunch

14:00-14:30 *Formal Verification at AMD*
Arthur Flatau, Matt Kaufmann, David F. Reed, David Russinoff, Eric Smith, and Rob Sumners (AMD)

14:30-15:00 *Verifying Processor Designs at the RTL Level*
Ken McMillan (Cadence)

15:00-16:00 Coffee and discussion

16:00-16:30 *An Introduction to Abstract Interpretation*
Nicolas Halbwachs (VERIMAG, Grenoble)

16:30-17:00 *Modular analysis of circuit description language by abstract interpretation*
Charles Hymans (École Polytechnique, Palaiseau)

Sunday, 7 April 2002

9:00-9:30 *Embedded Languages for Hardware Compilation*
Koen Claessen and Gordon Pace (Chalmers University of Technology and INRIA)

9:30-10:00 *Modeling systems with Streams in Daisy/The SchemEngine Project*
Steven D. Johnson (Indiana University)

10:00-10:30 *Functional design using behavioural and structural components*
Richard Sharp and Alan Mycroft (Cambridge University)

10:30-11:00 Coffee

11:00-11:30 *A Proof Engine Approach to Solving Combinational Design Automation Problems*
Gunnar Andersson, Per Bjesse, and Byron Cook (Prover Technology)

11:30-12:00 *State Abstraction Techniques for the Verification of Synchronous Circuits*
Yannis Bres, Gérard Berry, Amar Bouali and Ellen M. Sentovich (INRIA, Esterel Technologies and Cadence)

12:00-14:00 Lunch

14:00-14:30 *An approach to the introduction of formal validation in an asynchronous circuit design flow*
Dominique Borrione, Menouer Boubeker, Marc Renaudin, and Jean-Baptiste Rigaud (TIMA laboratory, Grenoble)

14:30-15:00 *Issues in multiprocessor memory consistency protocol design and verification*
Ritwik Bhattacharya and Ganesh Gopalakrishnan (University of Utah)

15:00-16:00 Coffee and discussion

16:00-16:30 *Specifying and verifying fault-tolerant hardware*
Scott Hazelhurst and Jean Arlat (University of the Witwatersrand and LAAS-CNRS)

16:30-17:00 *A Stream-Based Framework for Reasoning with STE and other LTL Verification Formalisms*
John O'Leary and Tom Melham (Intel and Glasgow University)

Abstract: Constraint-Based Verification

Authors: Carl Pixley (Synopsys Inc.) Ken Albin and John Havlicek (Motorola Inc.)

Introduction

Constraint-Based Verification, Yuan et al. [0], [1], Shimizu et al. [2], [3], Shiple et al. [4] has proven to be a practical approach to verification at the module, block and unit levels of hardware design and reused hierarchically at higher levels of design. The technique is to define a set of Boolean constraints, referencing any nets in the design or monitors (i.e., auxiliary deterministic finite state machines), to act as an "environment" for a Design Under Verification (DUV). Since constraints are not "forward looking", they can be used to generate inputs to the design. Stimulus generation can be done efficiently and on-the-fly during simulation. Since constraints do not involve fake random inputs (i.e., auxiliary free variables), they can also be used to monitor inputs to the design efficiently and during simulation. For formal verification, constraints can be used as a model of the environment of the design for the purpose of model checking, Kaufmann et al. [5].

Nice properties of a constraint-based verification system:

There is a duality between constraint-based simulation generators and monitors as well as between constraints as assumptions and as proof obligations. Boolean constraints can easily support both identities. This implies that constraints developed to be simulation generators for a DUV can easily become obligations for neighboring blocks during block-level verification and during integration. In addition, constraints developed to generate inputs for bus interface unit can be converted to monitors to verify the outputs of the same bus unit. This happened in the development of the Rapid-I/O bus interface module, exposing previously unknown bugs. Constraint-based monitors were also used to find bugs in a PCI bus protocol itself [2]. The multiple roles of constraints support reuse of information in a fundamental way. By contrast, conventional testbench stimulus drivers are usually discarded during integration verification. Constraints are normally delivered to the consumer of Intellectual Property (IP) to be used during IP integration.

Constraints can be developed incrementally and inexpensively as the design matures. In the earliest stages of design simple constraints can be used to animate the design almost effortlessly and waveforms can be observed. In our experience, designers have found bugs at the earliest stages of design, even before assertions (i.e., properties, checkers) were written. Also, no elaborate, expensive testbench is needed to perform constraint-based simulation. Therefore, designers can use constraints directly without the need for verification specialists. In addition, the development of constraints is amortized over time. There is no one-time, up front cost for developing constraints, as is common with testbench programs.

Of course as the design develops, the input constraints have to become accurate. This is accomplished by the generation of false negatives by either the simulation or formal engines. It is possible for constraints to be too "tight" in which case the constraints will be observed to fail when they are "flipped" to become checkers (i.e., obligations for neighboring blocks) during system integration. If constraints are too "loose," an expected property may fail in simulation or formal verification. An additional benefit of the constraint-based simulation approach is that constraints formally document the interfaces to DUV's in a machine-readable way.

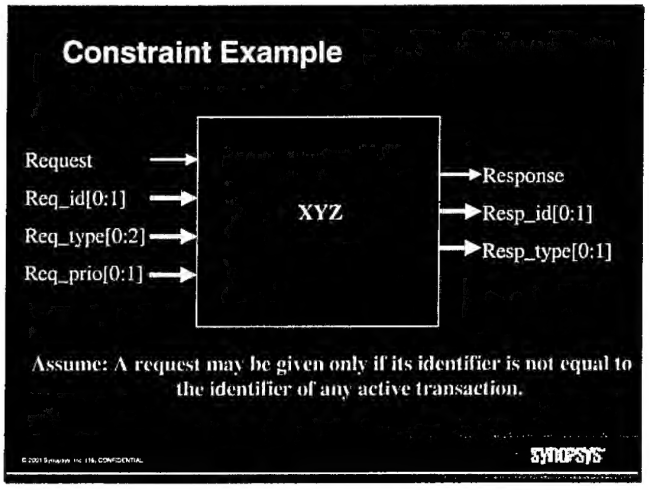
Constraint-based verification can be easily integrated into an existing simulation methodology, e.g., directed or directed-random simulation. This is an important practical consideration when trying to get a new methodology into existing design groups. Design groups are generally very conservative. They tend to stick with known methods. It is unlikely that they will adopt a new methodology if it is radically different from existing practice or if it is very expensive to integrate. The way constraint technology can be integrated into an existing simulation-based methodology is to develop constraints as checkers for existing, testbench generation programs. Then, when the constraints are mature they can flip to become generators. Of course, one of the added side benefits is that with constraints, one can formally model-check the DUV.

Constraint-based simulation is unexpectedly effective in finding bugs. "Corner cases" are found earlier. Empirical evidence for this is one design project in which constrained random simulation (C++ based) was used by a team of half a dozen people over a period of ten months finding only one design bug, while a single person used constraint-based simulation for only two months and found six (!) bugs confirmed by designers.

Constraint-based verification can be put directly in the hands of designers, rather than verification wizards, at the module, block and unit levels of design. This implies a much broader user-base for verification tools and technology. Furthermore, constraints certainly have all the benefits of an assertion-based methodology. For example, embedded constraints as checkers are left in the design as booby traps, which locate and isolate bugs to a particular site for easier diagnosis. It is worth noting that the language of constraints should be something familiar to designers, such as the Verilog expression language, in order to be adopted more readily. To be direct, the design community more readily adopts familiar languages.

Simple Constraint Examples:

Assume the following bus interface unit. Attached are two simple constraints on the interface to the unit: the environment is not allowed to a request for a request id when one is active and the type of a transaction must not be 5 or 7.




```
module xyz;

  /* Definitions Block */
```

```
    activate(id[0:1])[0:0]   = request  & (req_id == id) ;
    deactivate(id[0:1])[0:0] = response & (resp_id == id) ;
    active_next(id[0:1])[0:0] =
(
  deactivate(id) ? 1'b0   :
  activate(id)   ? 1'b1   :
                   active[id]
) ;

var active[0:15] =
{
  active_next(0),
  active_next(1),
  active_next(2),
  active_next(3),
} ;
```

Constraint: A request may be given only if its identifier is not equal to the identifier of any active transaction

```
constraint(request ? ~active[req_id] : 1'b1) ;
```

Constraint: The type of a request should never be 5 or 7.

```
constraint(request ? ((req_type != 5) & (req_type != 7)) : 1'b1) ;

endmodule
```

SimGen Constraint Generation

Simulation generation from constraints a la SimGen [2] works very simply. At compile time the user supplies a set of constraints to the SimGen compiler. The constraints are compiled either into a Verilog module [4] or into a C program [1] that runs during simulation. During simulation a user can give a set of biases to the runtime program to control the likelihood that an input bit will get set to 1 The user can also set an initial state using Verilog directives or can give the design a synchronizing sequence. At any point the user can call a SimGen task, which turns the simulation over to the SimGen executable. After that point SimGen generates simulations compliant with the constraints and biases every clock cycle.

There is a logical (and very real) possibility that the simulation will encounter a deadend state, i.e., a state for which there is no solution for the inputs. At that point the simulation will stop and the offending trace will be generated.

Bibliography

[0] J. Yuan, K. Shultz, C. Pixley, H. Miller, "SimGen: A Tool for Automatically Generating Simulation Environments from Constraints", ITC Workshop on Microprocessor Test and Verification, October 22-23, 1998

[1] J. Juan, K. Shultz, C. Pixley, H. Miller, A. Aziz, "Modeling Design Constraints and Biasing in Simulation Using BDDs", ICCAD 1999

[2] James H. Kukula and Thomas R. Shiple, "Building Circuits from Relations" CAV 2000

[3] K. Shimizu, D. L. Dill, and A. J. Hu. "Monitor-Based Formal Specification of PCI", FMCAD 2000, Austin, Texas.

[4] K. Shimizu, D. L. Dill, C-T. Chou, "A Specification Methodology by a Collection of Compact Properties as Applied to the Intel Itanium Processor Bus Protocol", CHARME 2001, Livingston, Scotland.

[5] Matt Kaufmann, A. Martin, C. Pixley, "Design Constraints in Symbolic Model Checking", CAV 1998: 477-487

Unifying Traditional and Formal Verification through Property Checking

Harry Foster
harry@verplex.com
Verplex Systems, Inc

Extended Abstract

In verification, there is no "silver bullet". However, verification methodologies can be improved with more effective techniques that include a combination of traditional simulation, semi-formal bug-hunting techniques, and formal property checking. The unifying factor, and necessary ingredient, behind successfully combining traditional with formal verification is *property specification* (combined with methodology considerations). This presentation explores property-checking techniques that were used to achieve functional closure (that is, ensured that a design met its functional specification, from RTL implementation to final layout) on a Hewlett-Packard highend server ASIC project. The presentation initially explores simulation and formal property checking used within a functional verification methodology. Then, the presentation will demonstrate how to combine formal property checking with logical equivalence checking. The combination of these techniques ensures that both semantic *and* logical consistency is preserved during design transformations.

Enhancing Functional Verification with Property Checking: Property specification continues to be problematic, partially due to the lack of a standard property language, but compounded by a lack of commercial tool support for specification-driven verification. Currently, Accellera (see www.accellera.org)—whose mission is to drive worldwide development and use of standards that enhance a language-based design automation process—is addressing the former. It is in the process of adopting a formal property language through the efforts of its Formal Verification Committee (see www.eda.org/vfv). Two of the formal property languages under consideration for standardization are the Motorola *CBV* and the IBM *Sugar*. These powerful and expressive formal property languages will enable engineers to:

- specify properties and constraints for formal analysis (for example, *property checking*)
- specify functional coverage models to measure the quality of simulation
- develop pseudo-random constraint-driven simulation environments derived from formal specifications [Yuan, et al. 1999]

When unifying traditional and formal verification, developing an effective methodology is equally as important as the property languages (and formal tools). That is, standardizing a property language, is not the entire solution. Recently, monitor-based methodologies have emerged as a mechanism for unifying traditional and formal verification (for example, *FoCs-Automatic Generation of Simulation Checkers from Formal Specification* [Abarbanel, et al. CAV 2000]). Other approaches include creating a protocol bus-monitor that examines an agent's output signals (as the monitor's input) and generates a Boolean $correct_i$ output signal, which is true when agent i is compliant to the specification (for example, *Monitor-Based Formal Specification of PCI* [Shinmizu et al. FMCAD 2000] and *A Specification Methodology by a Collection of Compact Properties as Applied to the Intel Itanium Processor Bus Protocol* [Shimizu et al. CHARME 2001]).

In this presentation, the author presents his own experience with a monitor-based technique for specifying RT-level implementation properties, using the *Open Verification Library* set of assertion monitors [Foster and Coelho HDLCON 2001]. Data will be presented comparing two similar highend server ASIC design projects—one with a monitor-based approach to specifying RTL implementation assertions and one without monitors. The monitor-based methodology consisted of over 4000 RT-level implementation assertions and over 8000 RT-level functional coverage points (using the same monitor-based form of specification). The monitor-based approach significantly reduced simulation debug time compared to the non-monitor-based approach (15 minutes, on average, compared to hours). Furthermore, the methodology demonstrated how the same form of specification could be leveraged across multiple verification process (for example, direct/random simulation, semi-formal verification, and formal verification). The monitor specifications succeeded in identifying 85% of all recorded bugs (15% were identified by other techniques).

Enhancing Equivalence Checking with Property Checking: Transformation verification, using formal combinatorial equivalence checking, has become mainstream for design projects over the last few years. Although the techniques are extremely useful for ensuring logical consistency, thus minimizing the need for gate-level simulation, it is not a complete solution for verifying equivalence. Designers must be aware of a class of functional bugs that cannot be demonstrated on the RTL model due to optimistic behavior of X in RTL simulation. Furthermore, many synthesis pragmas, such as *full_case* and *parallel_case* create semantic inconsistency (and potential synthesis bugs) between the pre-synthesis and post-synthesis designs. Gate-level simulation is required to identify the problem—yet the two circuits will prove logically equivalent.

Although it would be desirable to enforce a restricted coding style to prevent semantic inconsistencies [Bening and Foster 2001], this is not always possible—particularly when integrating in IP or coding to solve timing issues. However, property checking can be applied to ensure safe usage of X and synthesis pragmas within the RTL.

The following Verilog code illustrates one form of semantic consistency problem associated with a *full_case* synthesis pragma.

```
module mux (a,b,s,q);
  output      q;
  input       a, b;
  input [1:0] s;
  reg         q;
  always @(a or b or s) begin
    case (s) //rtl_synthesis full_case
     2'b01: q = a;
     2'b10: q = b;
    endcase
  end
endmodule
```

Pre- versus post-synthesis semantic inconsistency occurs when an error in the logic driving the *"s"* variable generates a value other than the alternatives 2'b01 and 2'b10. For the pre-synthesis simulation behavior, if *"s"* assumes an illegal value (for example, 2'b11, 2'b00, or 2'bXX), then the *"q"* variable incorrectly behaves as a latch in RTL simulation—holding its previous valid value. However, the post-synthesis model contains no latch. This causes a prospective pre- versus post-synthesis simulation difference, which means that a functional bug could be missed during RTL simulation; whereas, this same bug could be uncovered during gate-level simulation (for logically equivalent circuits). Of course, our goal is to minimize the gate-level simulation bottleneck. Hence, formal equivalence checking is only a partial solution; since it doesn't

ensure semantic consistency. However, formal property checking helps us in these cases by verifying that the condition resulting in an X assignment would never occur (or would never be observed). Furthermore, property checking ensures that conditions specified by a synthesis pragma are not violated. For example, on the Hewlett-Packard highend server project, synthesis pragma violations were identified on the IP and corrected prior to synthesis using formal techniques.

Conclusion: Using a systematic monitor-based library of checkers to verify general design RT-level implementation properties, as well as validate semantic consistency during equivalence checking, has proven beneficial in unifying traditional and formal verification on actual design projects. Time-to-market reduction has been clearly demonstrated through reducing the debug time associated with traditional verification and identifying very complex bugs using semi-formal and formal verification.

Thoughts for academia: Based on experience in trying to integrate property checking techniques into both a traditional and formal verification flow, future research is desirable in the area of coverage associated with formal proofs (particularly related to bounded proofs). And, creating metrics that articulate coverage in a form that the designer or traditional verification engineer can relate to is also needed (for example; path coverage, toggle coverage, line coverage, etc.). Finally, hierarchical partitioning of RT-level proofs, while automatically deriving sub-block constraints based on witness generation, would be an interesting and potentially beneficial area of research for automation.

References:

[Abarbanel, et al. CAV 2000] Y. Abarbanel, I. Beer, L. Glushovsky, S. Keidar, Y. Wolfsthal, "FoCs: Automatic Generation of Simulation Checkers from Formal Specifications," *Proceedings of the Computer-Aided Conference* (CAV), pp. 538-542, 2000.

[Bening and Foster 2001] L. Bening, H. Foster, *Principles of Verifiable RTL Design*, Kluwer Academic Publishers, 2001.

[Foster and Coelho HDLCON 2001] H. Foster, C. Coelho, "Assertions Targeting A Diverse Set of Verification Tools," *Proceedings of the 10-th Annual International HDL Conference*, March, 2001.

[Shinmizu et al. FMCAD 2000] K. Shimizu, D. Dill, A. Hu, "Monitor-Based Formal Specification of PCI," *Proceedings of the Third International Conference on Formal Methods in Computer-Aided Design,* pp. 335-353, November 2000.

[Shimizu et al. CHARME 2001] K. Shimizu, D. Dill, C-T Chou, "A Specification Methodology by a Collection of Compact Properties as Applied to the Intel Itanium Processor Bus Protocol," In *CHARME'00*, Springer Verlag, pp. 340-354, 2001.

[Yuan, et al. 1999] J. Yuan, K. Shultz, C. Pixley, H. Miller, A. Aziz, "Modeling Design Constraints and Biasing in Simulation Using BDDs, " *Proceedings of the IEEE International Conference on Computer Aided Design*, pp. 584-589, November 1999

Array Verification at IBM Research

Warren A. Hunt, Jr.

IBM Austin Research Laboratory
11501 Burnet Road, MS 904-6H014
Austin, TX 78758

E-mail: whunt@austin.ibm.com
TEL: +1 512 838 9724
FAX: +1 512 838 4036

Array Verification at IBM Research

Warren A. Hunt, Jr.

IBM Austin Research Laboratory
11501 Burnet Road, MS 904-6H014
Austin, TX 78758

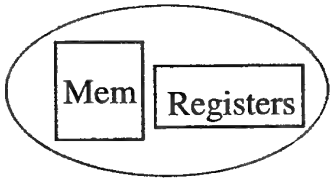
E-mail: whunt@austin.ibm.com
TEL: +1 512 838 9724
FAX: +1 512 838 4036

Overview of the Talk

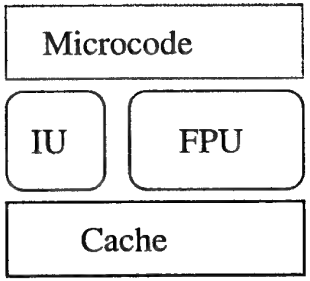
- Levels of abstraction
- Array Verification
- Example Array Verification
- Hierarchical Extraction
- Commercial Tools

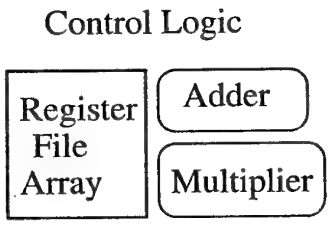
Levels of Abstraction

ISA

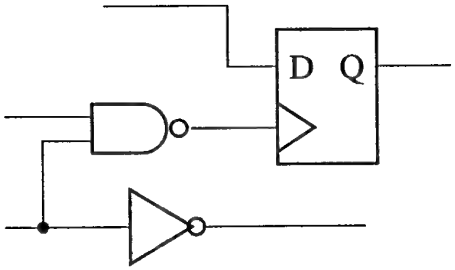


Microarchitecture

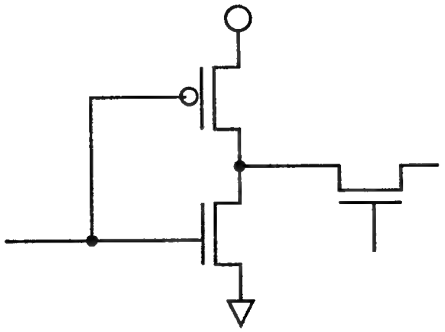




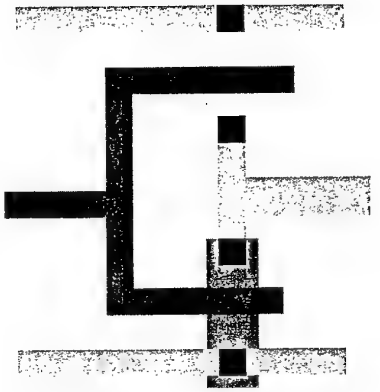
Netlist



Schematic



Layout



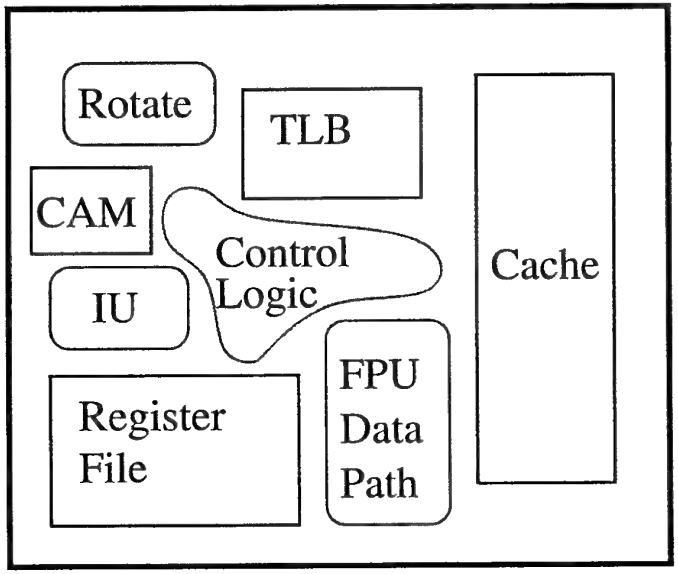
- Different specification languages are used at different levels.

The Verification Problem

- Different specification languages are used at each level.
  - ISA: C, C++ models
  - Architecture: Drawings, Charts, Graphs, Natural Language
  - Microarchitectures: More diagrams, charts, etc.
  - Register-transfer: VHDL, Verilog
  - Netlist: VHDL, Verilog
  - Transistor Schematic: "Stick diagrams"
  - Layout: Colored Polygons
- The Size
  - ISA models: hundreds of pages
  - RTL models: thousands of pages
  - Netlist models: millions of pages

Array Verification

- Problem: does the transistor-level design correctly implement an array?



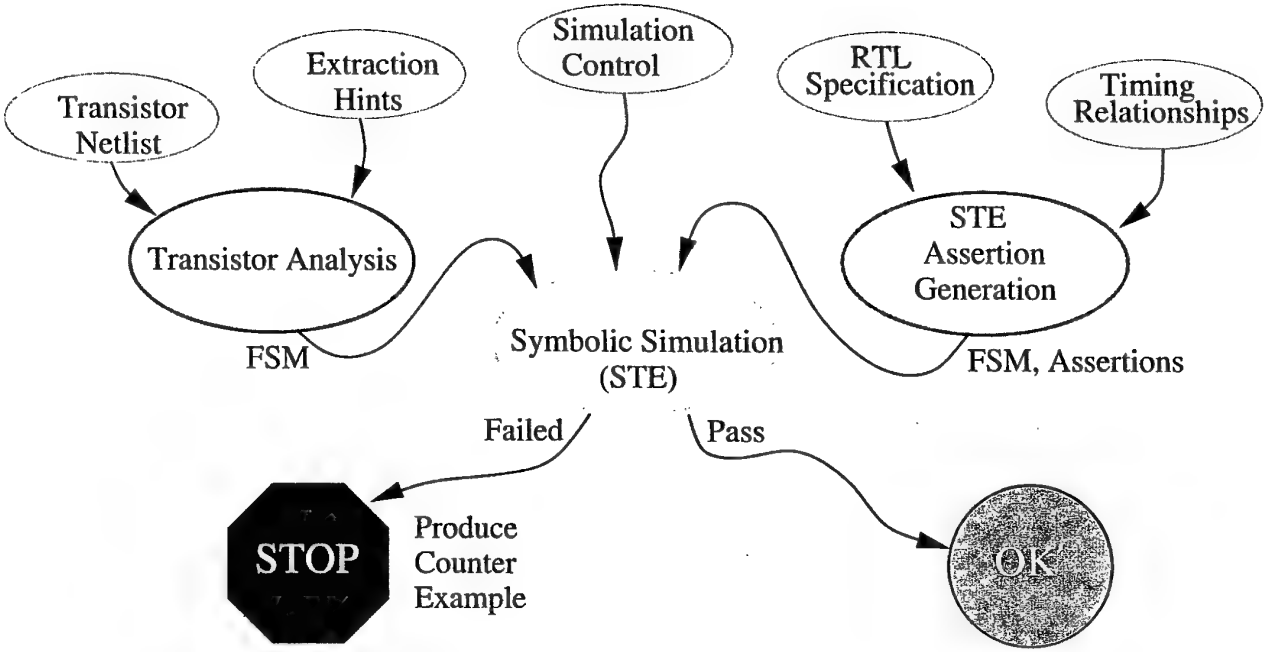
RTL Design Description

```
reg1 <= b_bus + a_bus;
n = max( a, b );
cache[n] <= mem_bus;
enable = a & cntl[3];
reg[i] <= sel( enable, i_bus, data_in);
```

- Important? Yes, more than 50% of transistors are for memory arrays.
- We extended a solution proposed by Pandey.
  - Transistor-level designs are converted into cooperating FSMs.
  - Proper stimulus environments are created.
  - RTL specifications are compared to FSMs using STE.

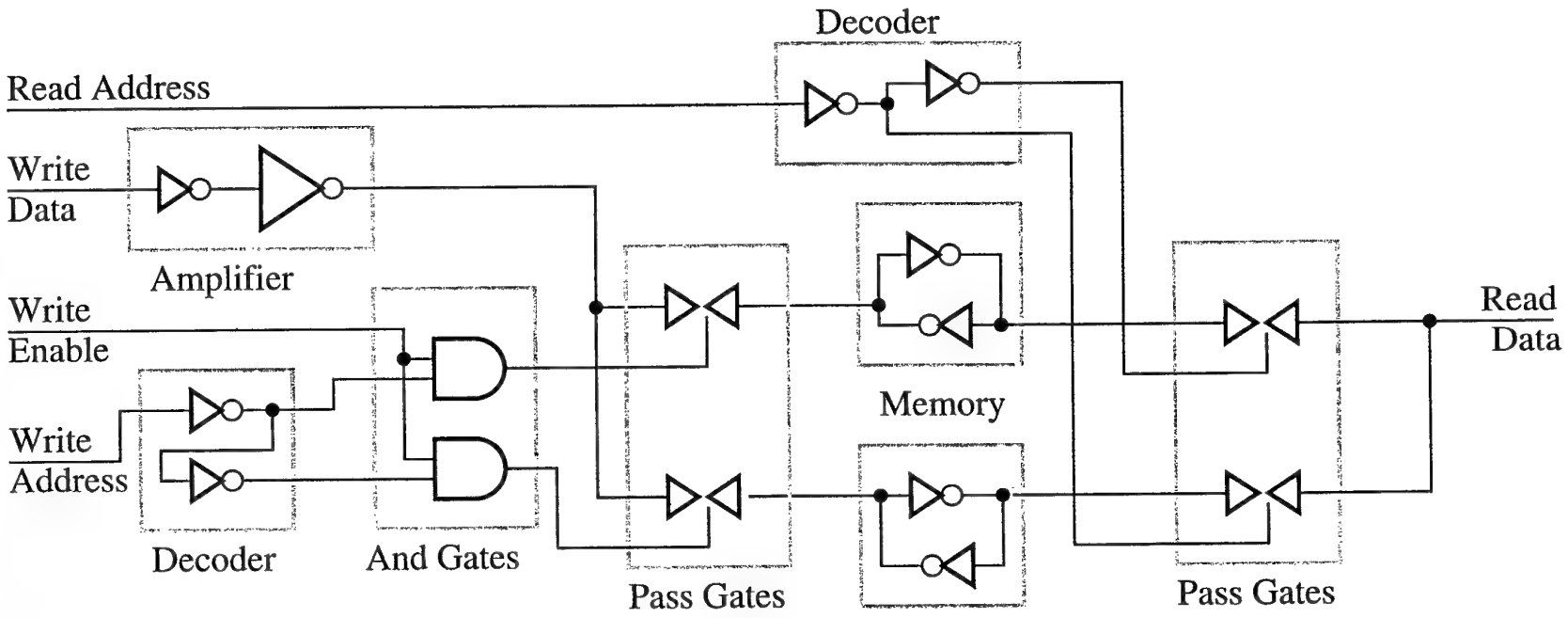
Verisym: An Array Verification Tool

- Verisym is composed of four parts:
  - Transistor Extractor
  - Symbolic Trajector Evaluator
  - Netlist Management and Coverters
  - User Interface



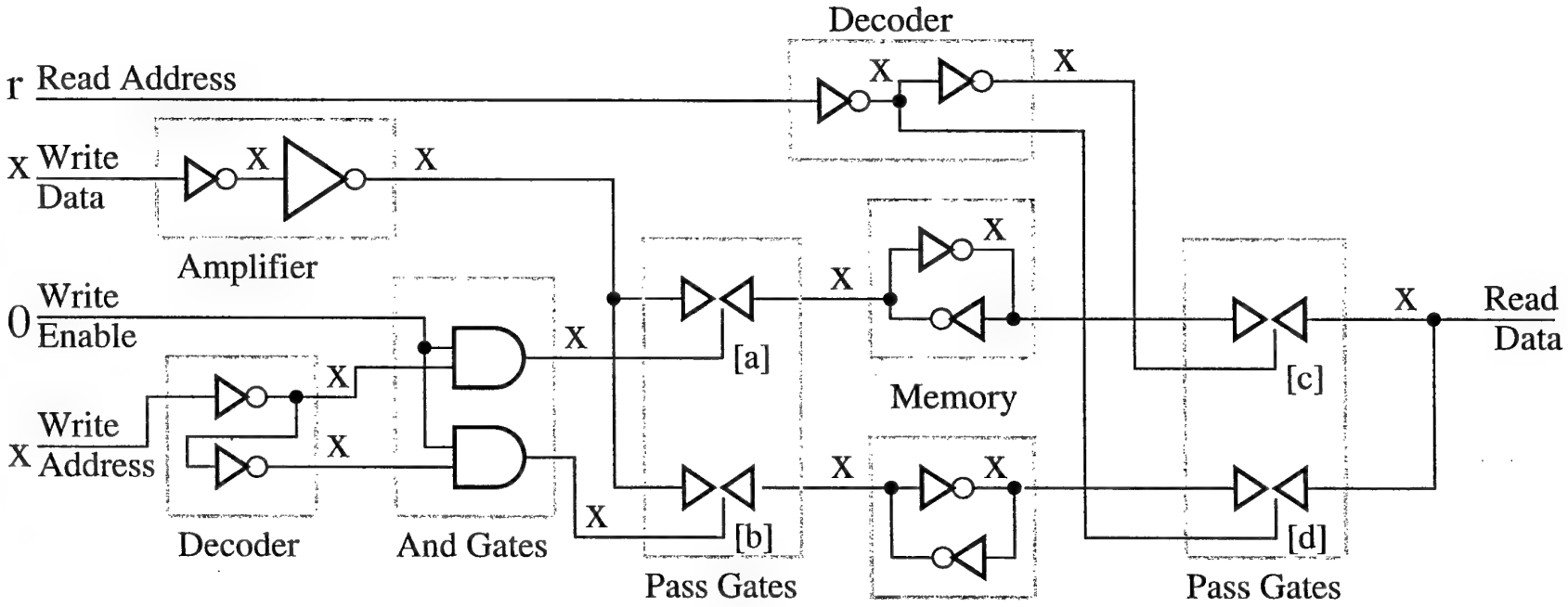
Example Array Verification

- To see how an array works, here is a simplified array design.



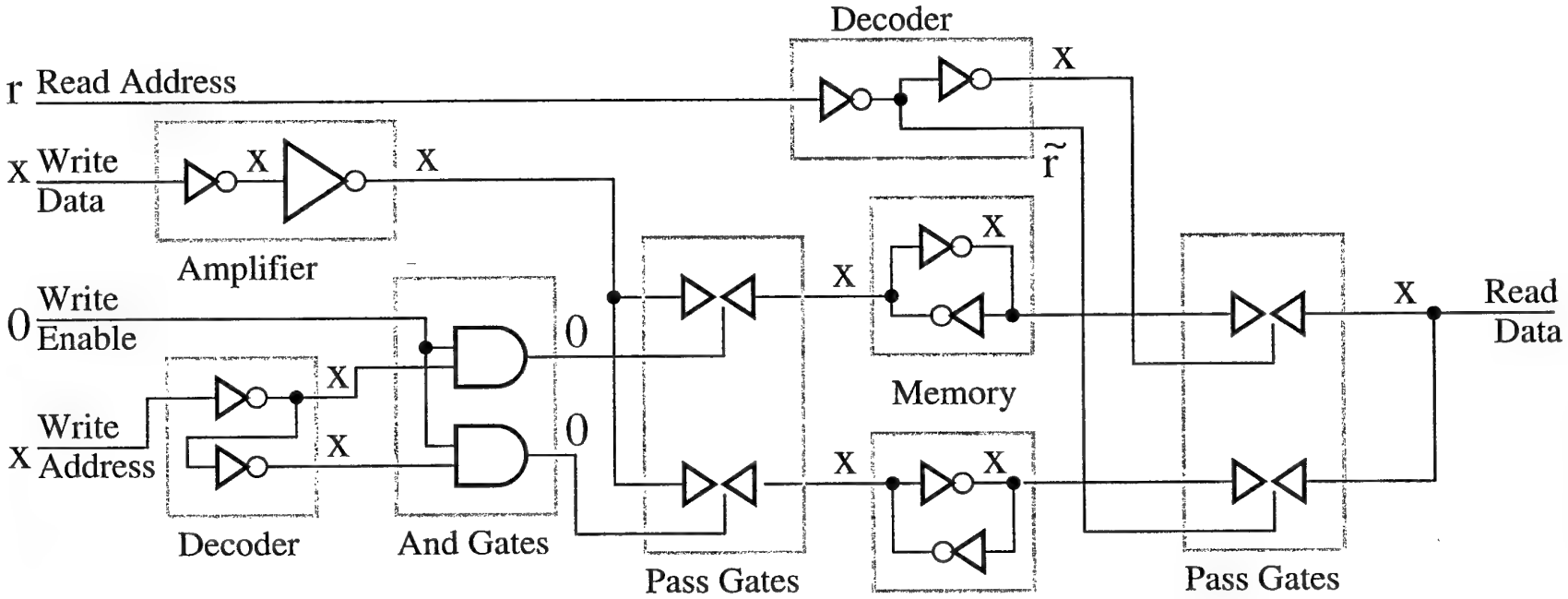
- Transistors have been simplified to gates for presentation purposes.
  - **Pass Gates** are bi-directional.
  - The **Amplifier** output strength is large.
  - The **Memory** elements are reinforcing.

Example Array Verification, Initialize



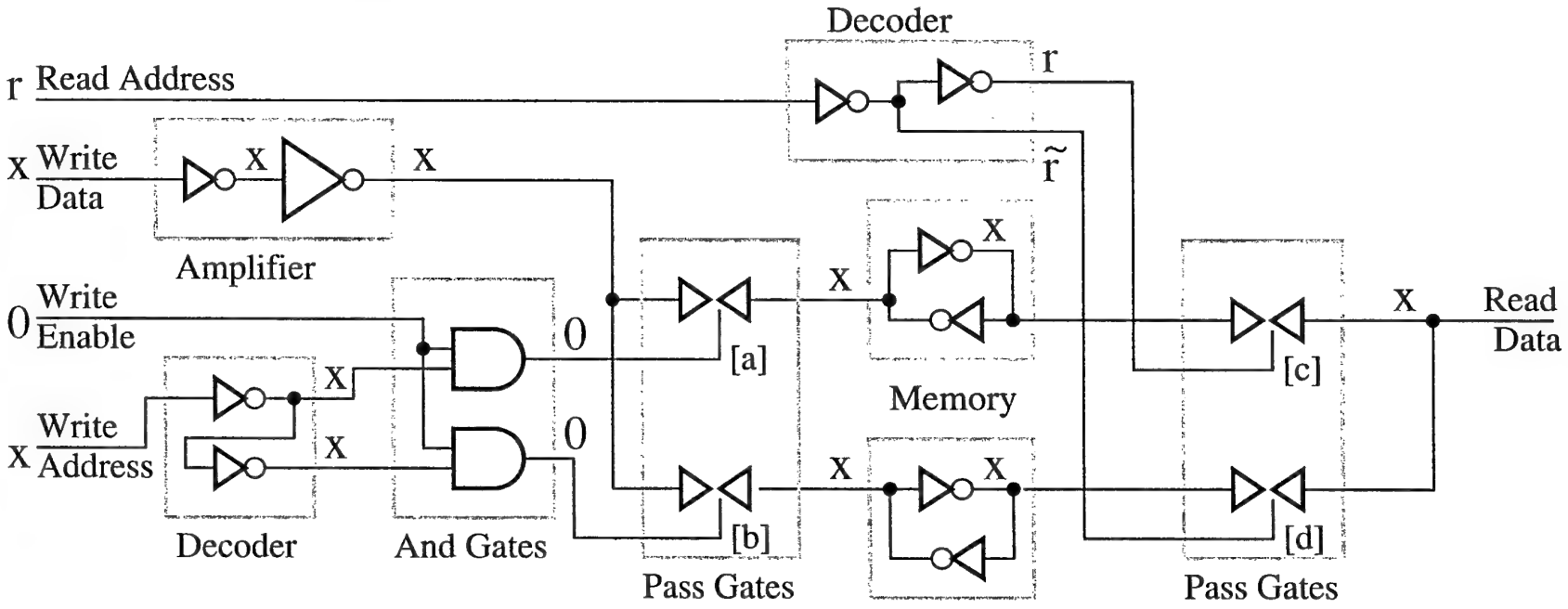
- Goal: To set the Pass Gate control signals to stable values.
- In terms of nodes: [a] = 0; [b] = 0; [c] = not([d]);

Example Array Verification, Initialize



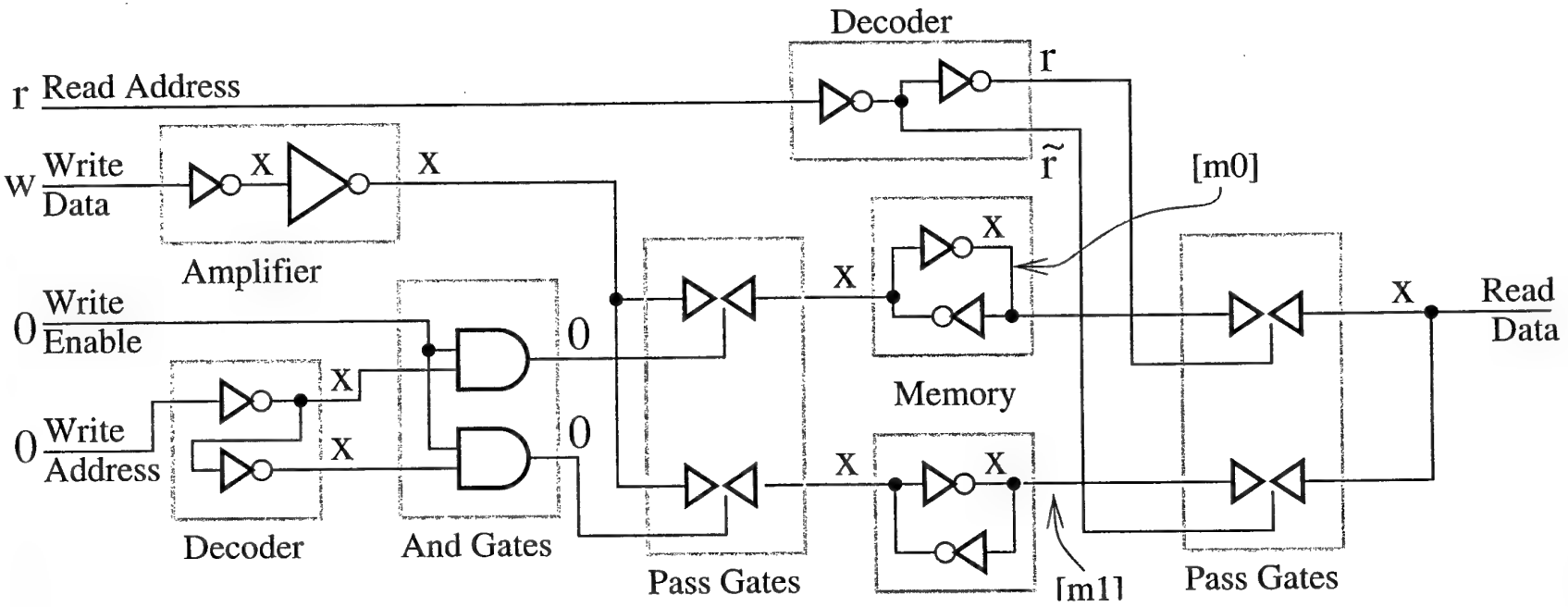
- The input Pass Gate control signals are disabled
- The two output Pass Gate control signals need to be inverted.

Example Array Verification, Initialized, Quiescent



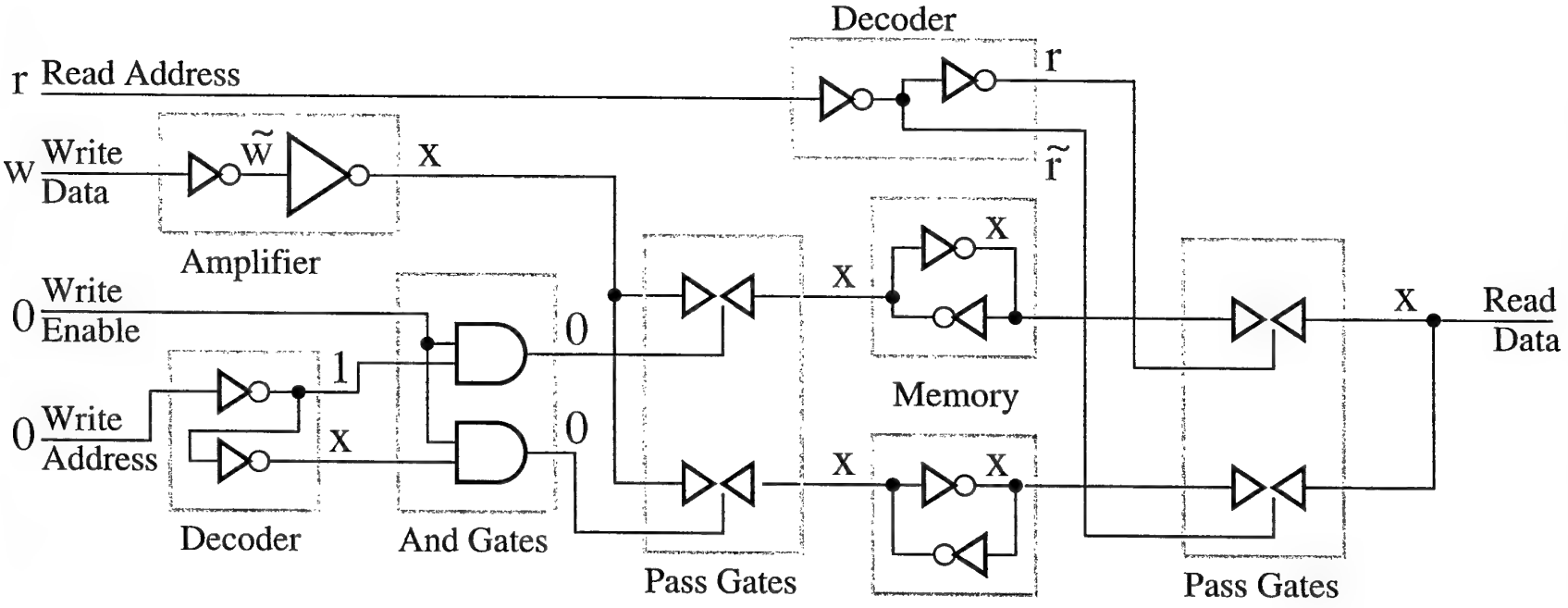
Time		i0	i1	i2
Antecedents	Read Address	r	r	r
	Write Data			
	Write Enable	0	0	0
	Write Address			
Consequents	[a]			0
	[b]			0
	[c]			not([d])
	[d]			not([c])

Example Array Verification, Write



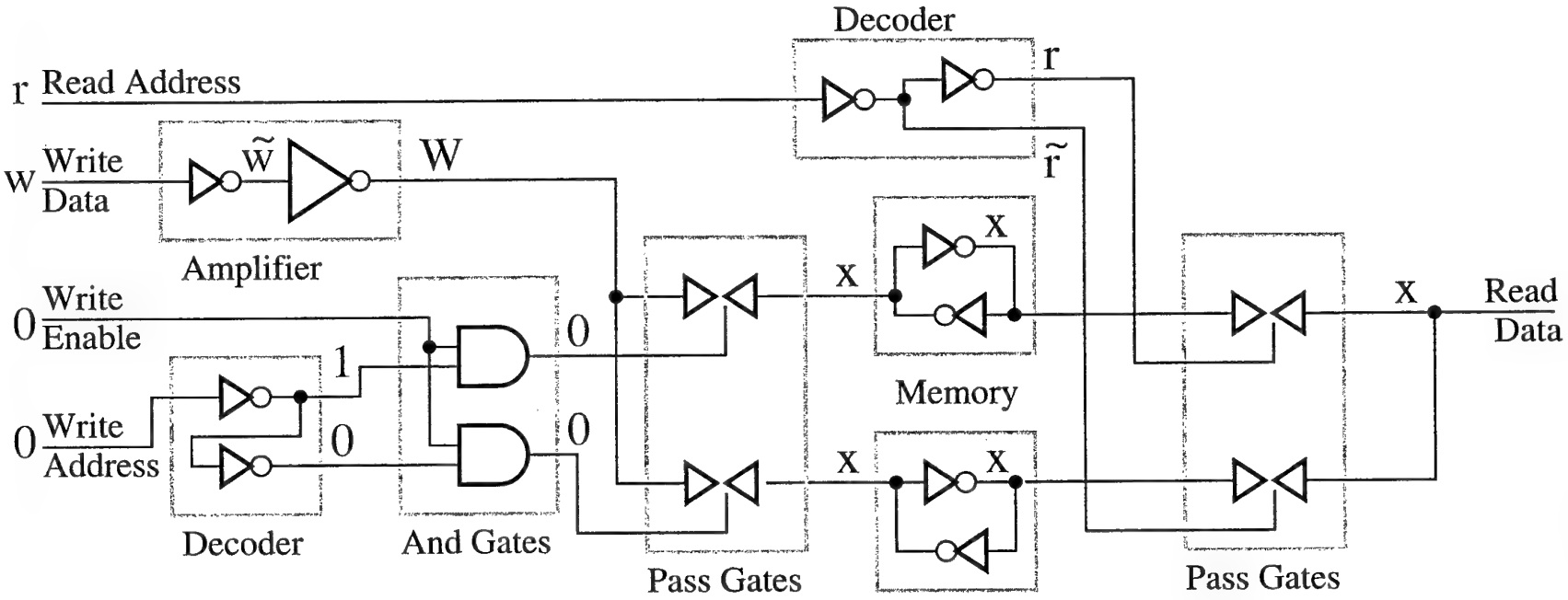
- Goal: [m0] = $\tilde{w}$

Example Array Verification, Write

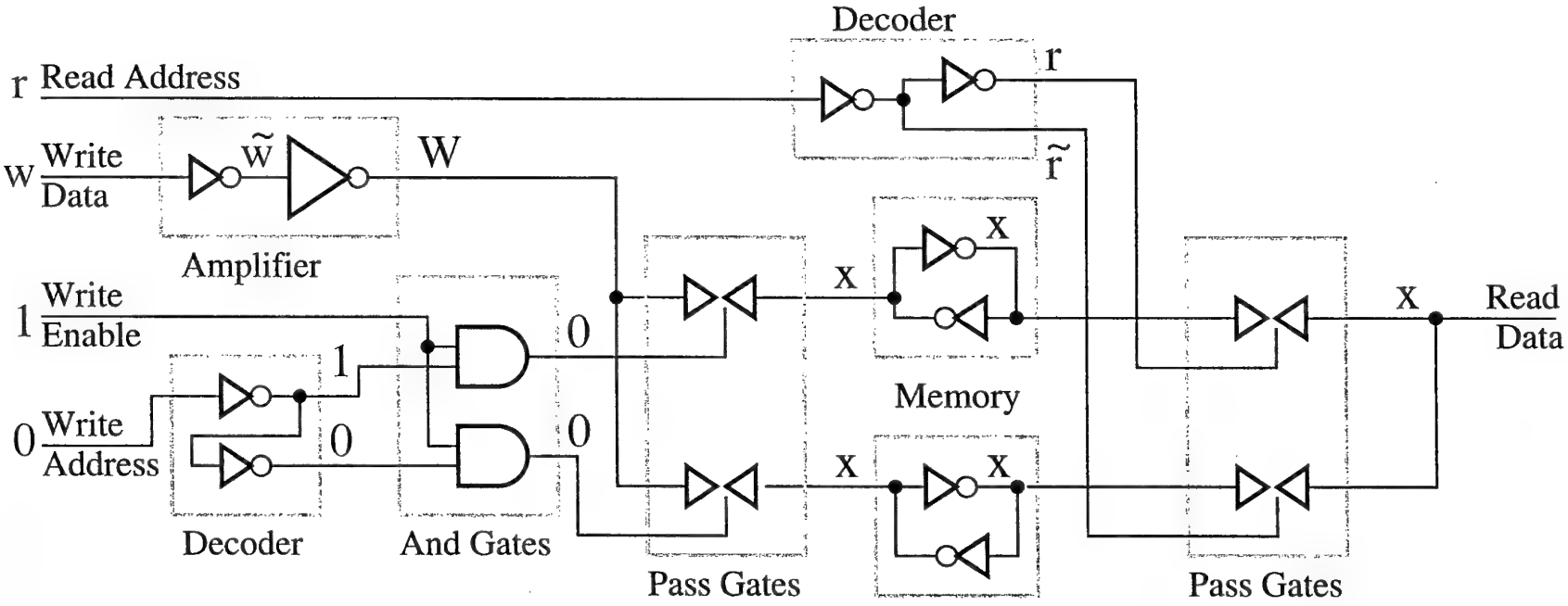


- Setup write data and write address.

Example Array Verification, Write

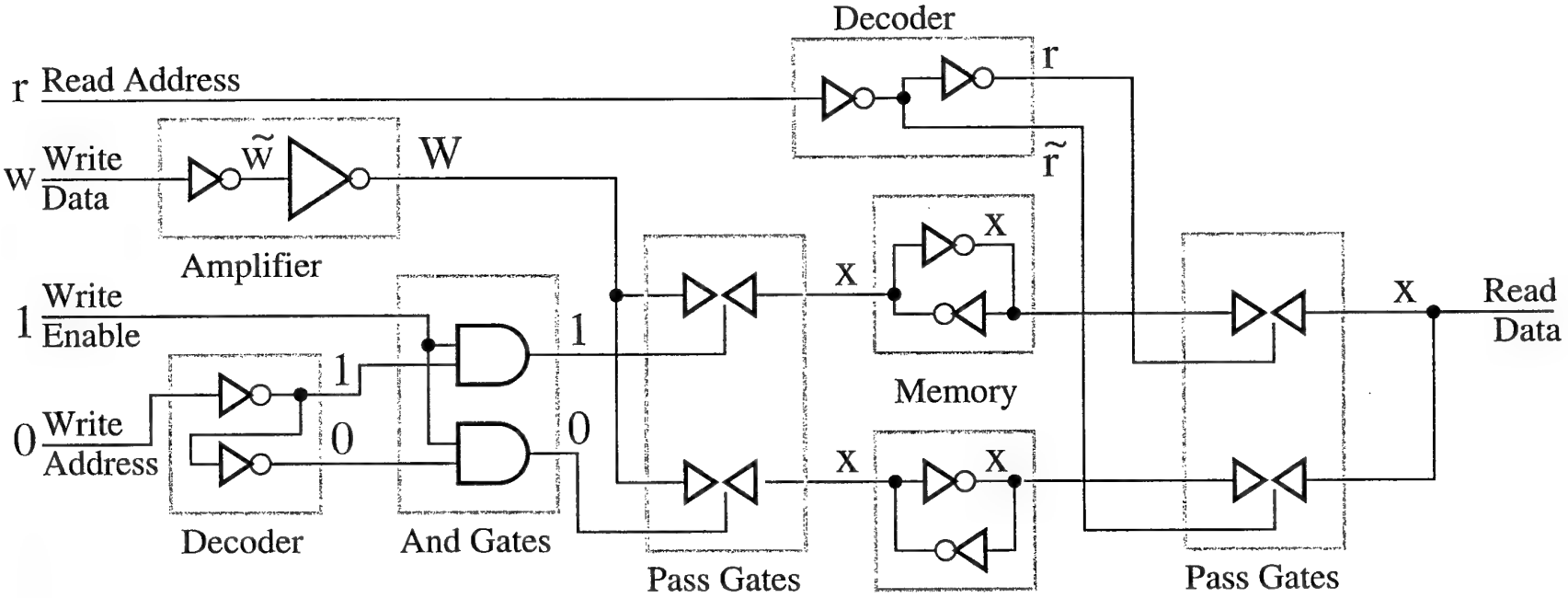


Example Array Verification, Write

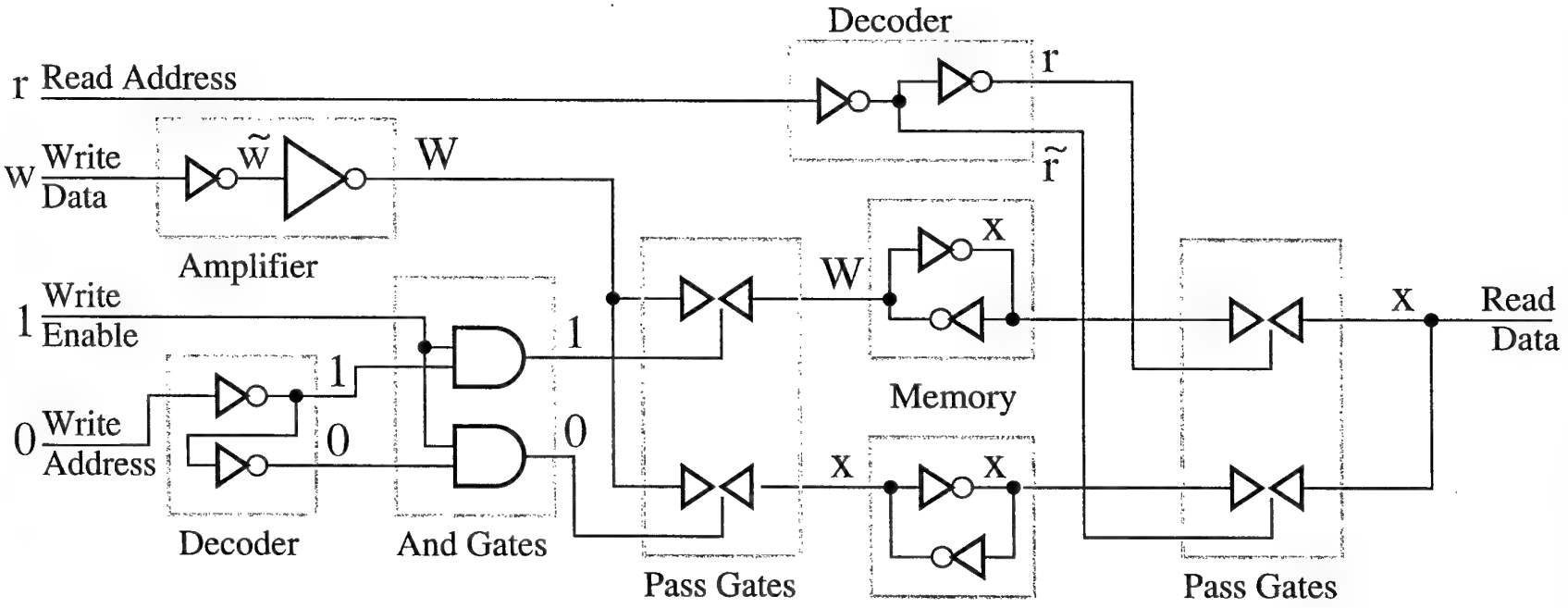


- Initiate write enable.

Example Array Verification, Write

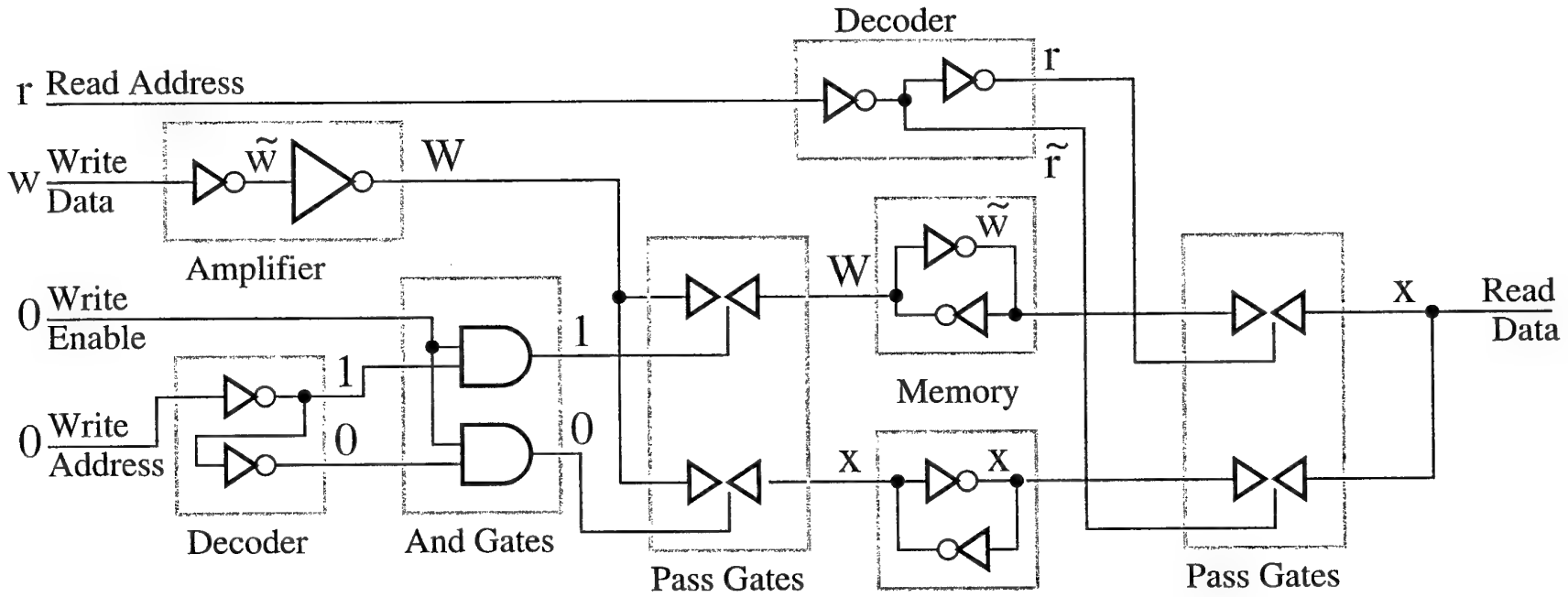


Example Array Verification, Write



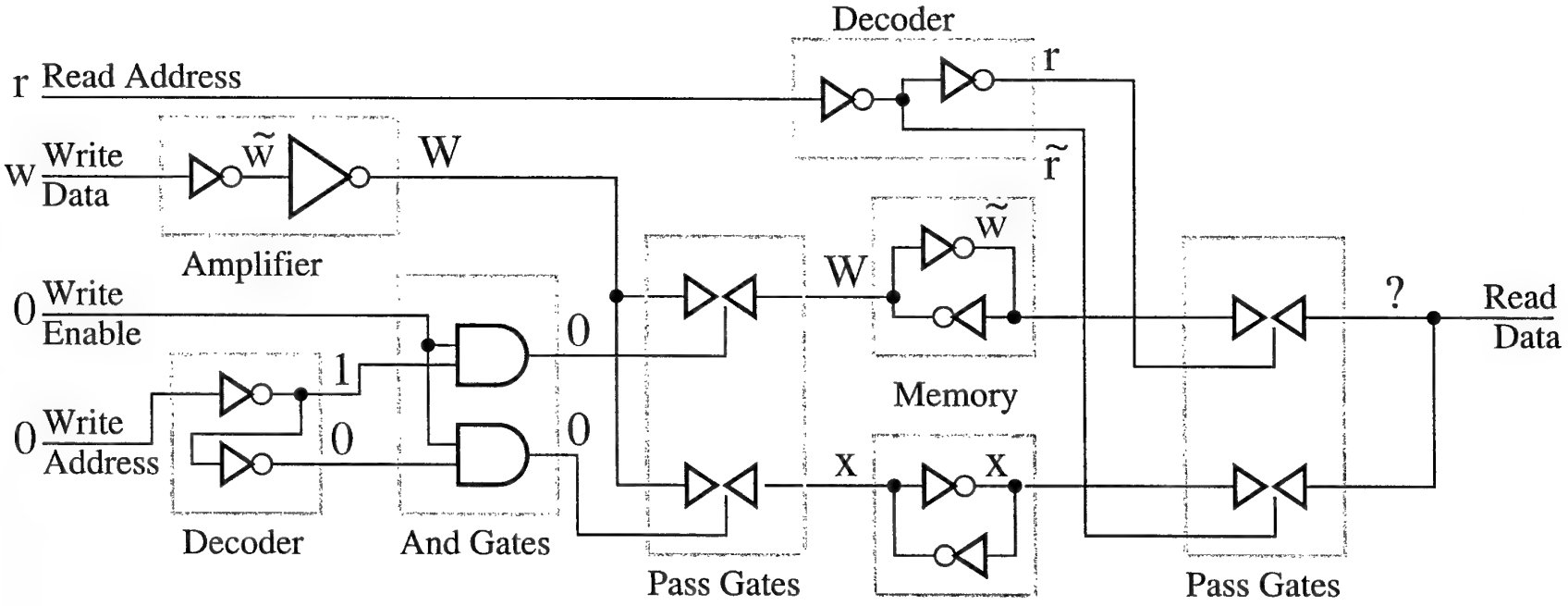
- "Overpower" inverter in memory cell.

Example Array Verification, Write

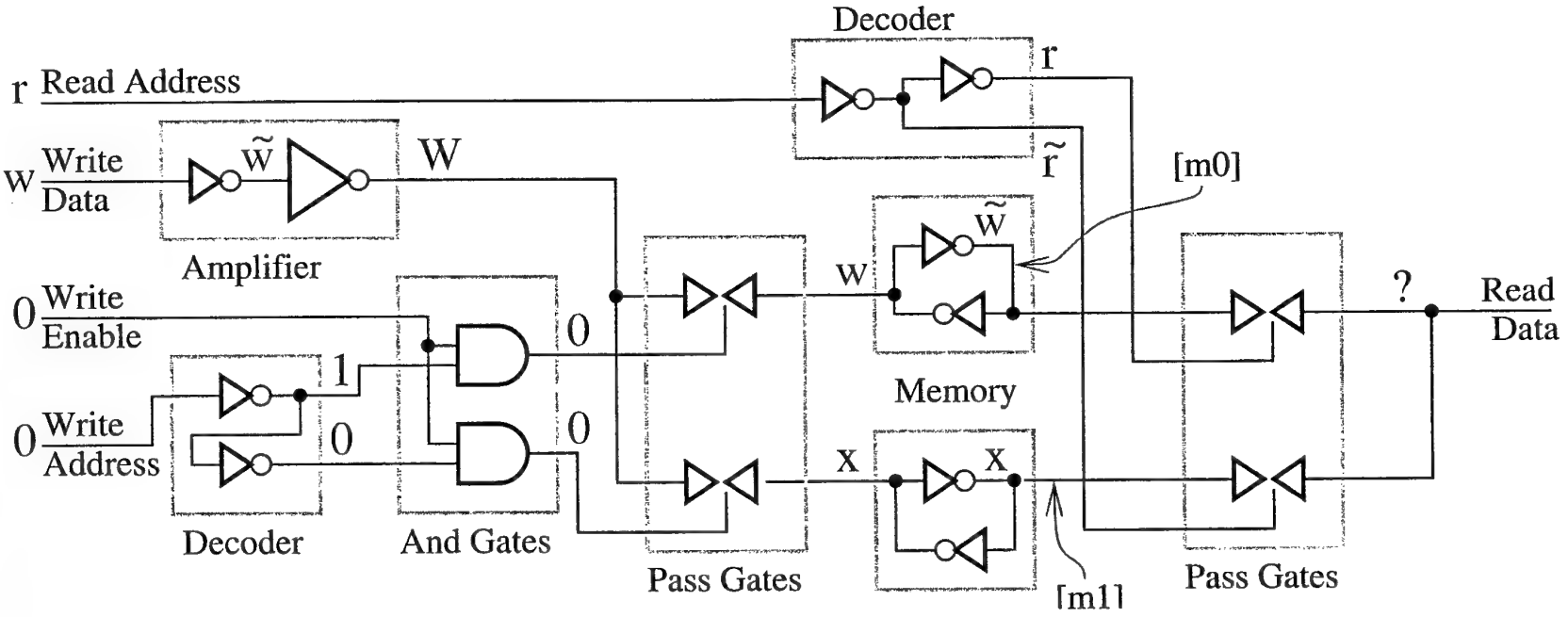


- Shutoff write enable.

Example Array Verification, Write

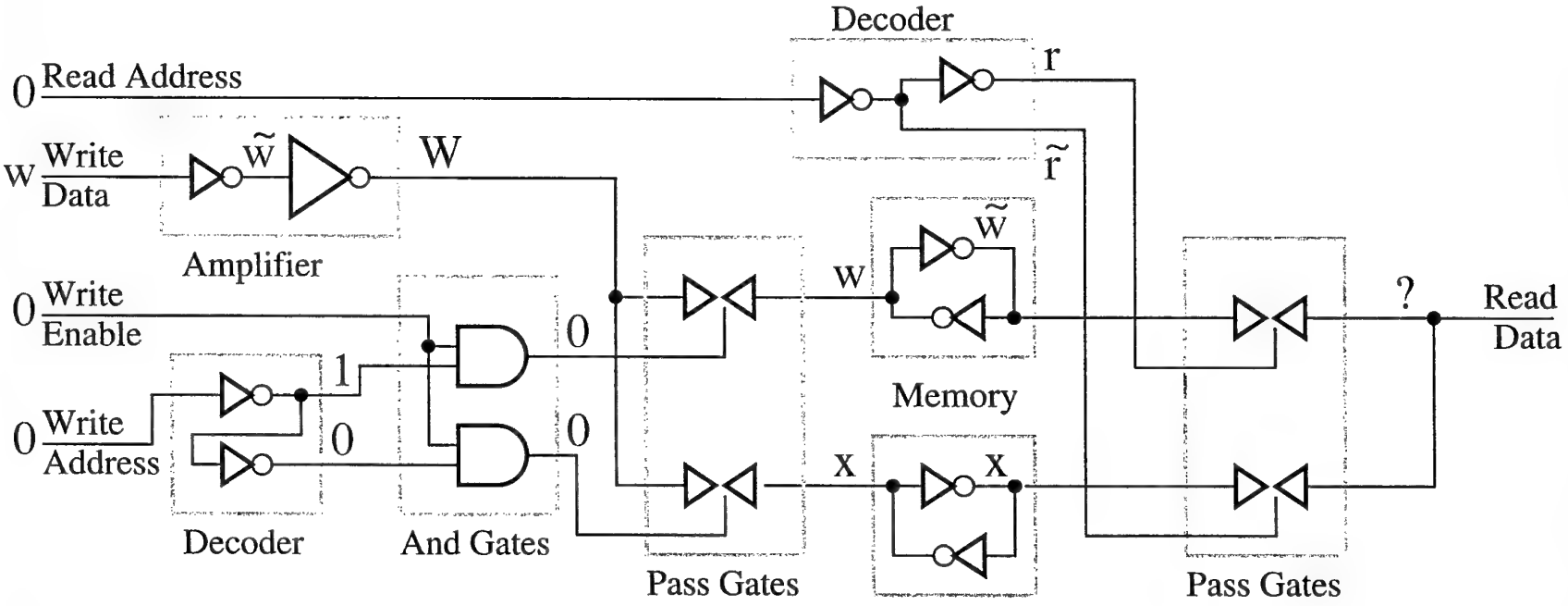


Example Array Verification, Write, Quiescent



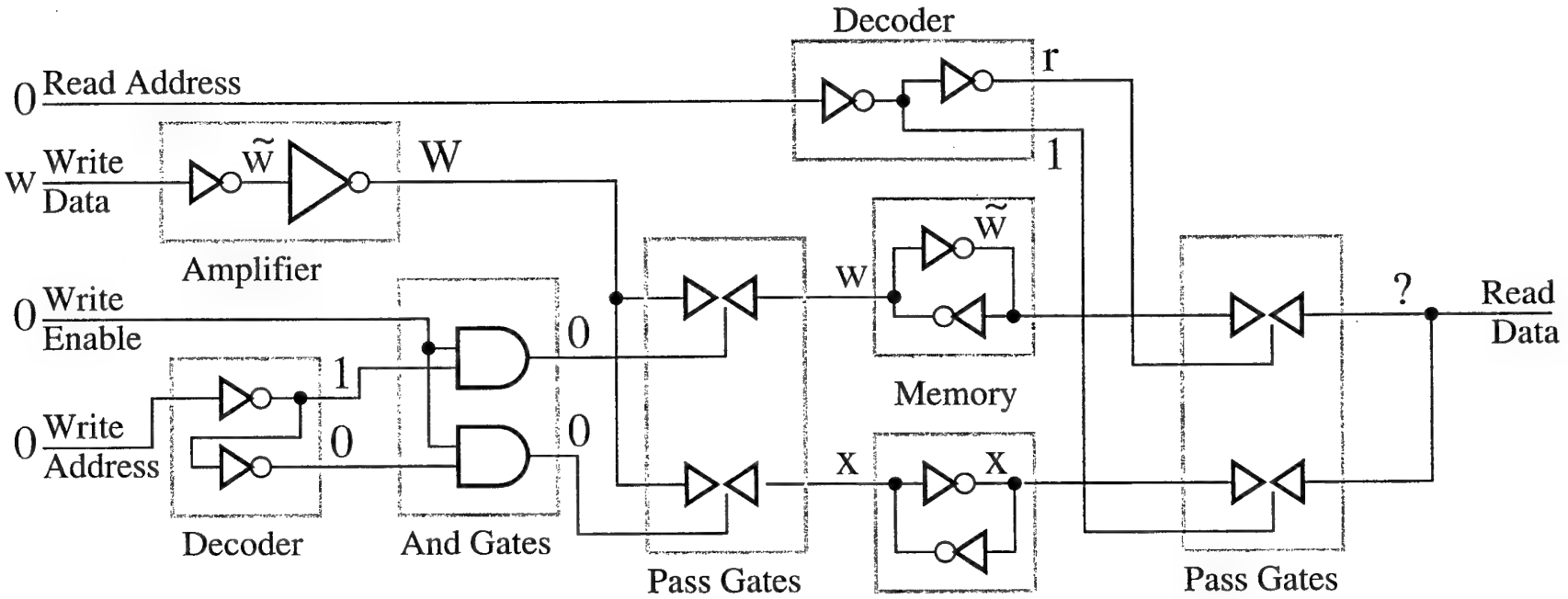
Time		w0	w1	w2	w3	w4	w5	w6	w7	w8
Antecedents	Read Address	r	r	r	r	r	r	r	r	r
	Write Data	w	w	w	w	w	w	w	w	w
	Write Enable	0	0	0	1	1	0	0	0	0
	Write Address	0	0	0	0	0	0	0	0	0
Consequents	Memory Bit 0 - [m0]								$\tilde{w}$	$\tilde{w}$
	Memory Bit 1 - [m1]									
	Read Data									

Example Array Verification, Read

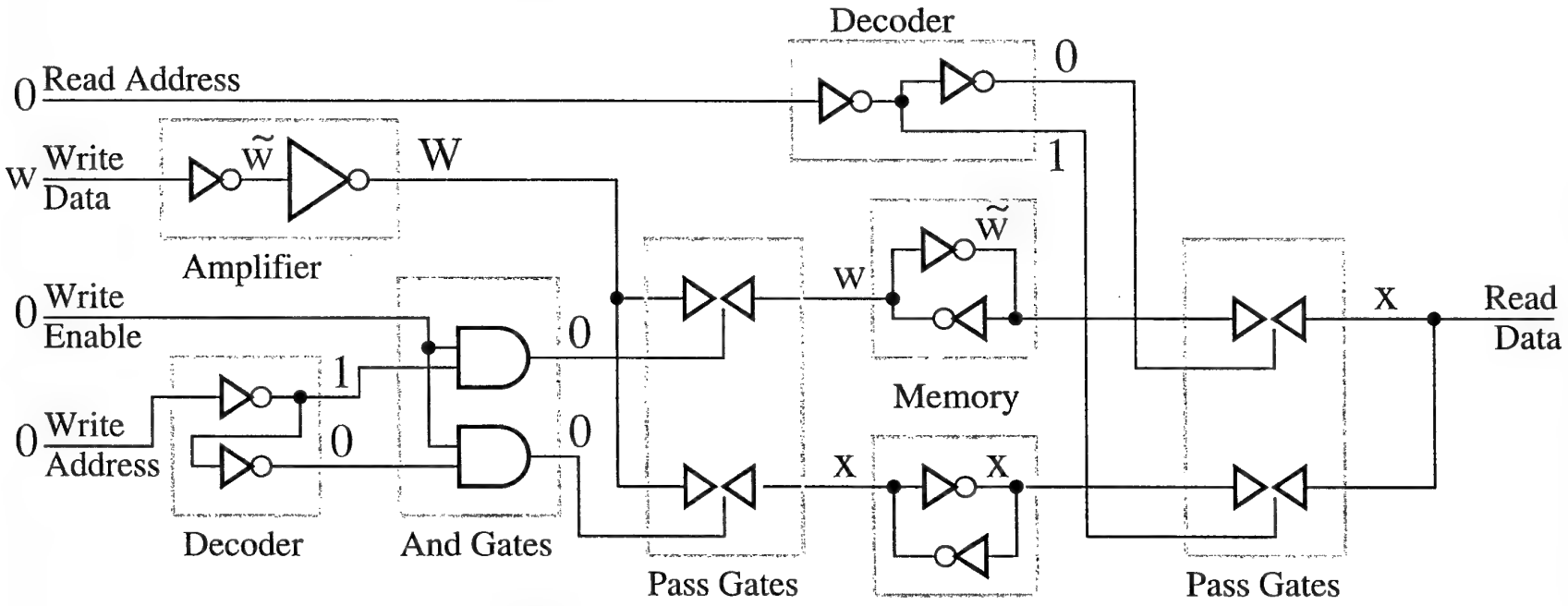


- Read memory location zero.

Example Array Verification, Read



Example Array Verification, Read



Time		r0	r1	r2
Antecedents	Read Address	0	0	0
	Write Data			
	Write Enable	0	0	0
	Write Address			
Consequents	Memory Bit 0			
	Memory Bit 1			
	Read Data			$\tilde{w}$

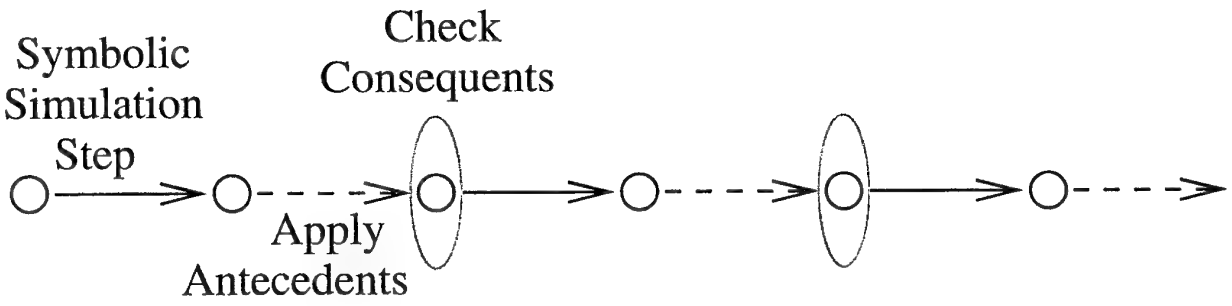
- Error! We have a design flaw. This type of error is quite common.

Array Verification Theorems

- Array theorems are expressed as STE assertion/consequent pairs.

$$A_t \Longrightarrow C_t$$

- Assertions and consequents are specified as equations.
  - For each step, the antecedents specify environmental conditions
  - And for each step, the consequents specify the expected results

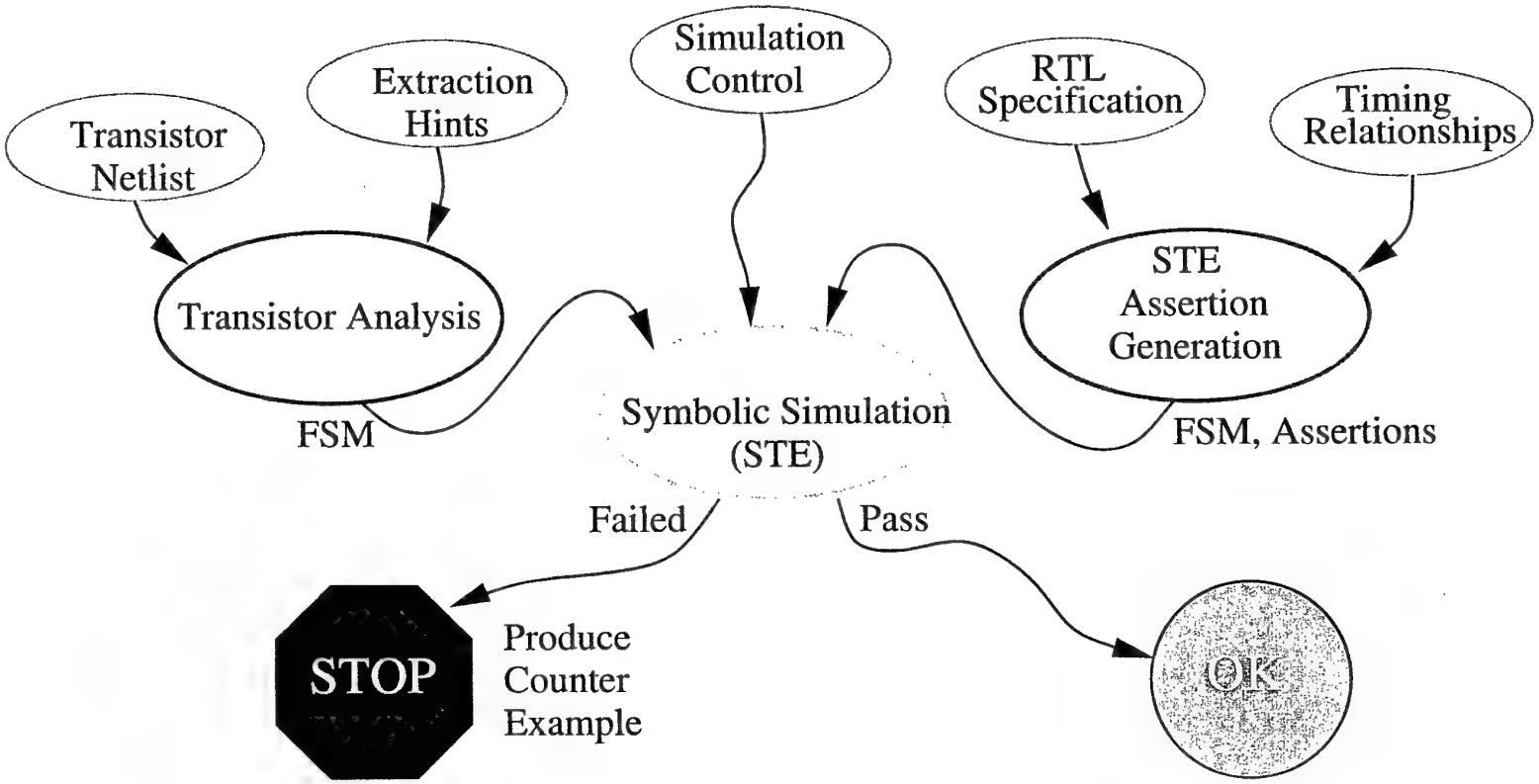


- For some initializations proofs, we need two state bits for each memory bit.
- During symbolic simulation the states can become very large. We have checked machines with tens of thousands of states.

Hierarchical Circuit Extraction

- Arrays are regular.
  - Many repeated structures.
  - Very large channel connected components (CCCs).
- Extraction accuracy is critical – built with ANAMOS model.
  - Unit-delay model.
  - Technology dependent strength analysis.
  - Timing and strength overrides supported.
- Hierarchical extraction improves performance.
  - Permits symmetry identification.
  - Eases association of external names to wire names.

Verisym: Array Verification by Extraction, STE



- System controlled from standard IBM tool interface.
  - Hierarchical transistor extractor – a first.
  - Automated conversion of RTL into STE assertions/consequents.
  - Industrial capacity.

What Makes Array Verification Difficult

- It may appear that array verification is straightforward, but:
  - size of arrays, multiport arrays,
  - CAMs, embedded logic,
  - simultaneous reading and writing,
  - size of channel connected components,
  - transistor size sorting, and
  - even finding the corresponding RTL description.
- Array theorems necessarily contain a complete design description.
  - Arrays have 100,000s of transistors; translates into very large FSMs.
  - Array specifications may be 10s of pages.
- Verification performance:
  - Extraction accuracy critical, speed less important,
  - The execution speed of our STE engine affects productivity, and
  - Regression suite verification a must.

Commercial Tools Require Substantial Effort

- To create a commercially viable tool required several iterations:
  - I wrote a demonstration tool written in late 1998
  - Alpha tool written by Will Adams, Damir Jamsek, and me in 1999.
  - Beta tool completed this year additionally involving Wendy Belluomini and Eric Mercer.
- Not such deep science, but deep engineering and methodology issues.
  - Integrating even straightforward hardware verification technology into IBM's tool flow requires tremendous effort.
  - In the last six months, a great deal of additional effort required from tools group for integration to IBM's tool flow.
- The real issue is size. Algorithms, file formats, names, etc., must be carefully considered.

Lava: An Embedded Language for Structural Hardware Design

Koen Claessen, Mary Sheeran, and Satnam Singh
{koen,ms}@cs.chalmers.se, Satnam.Singh@xilinx.com

Introduction

Lava is a tool to assist circuit designers in specifying, designing, verifying and implementing hardware. It is realised as an embedded language – a collection of libraries written in an already existing language, called the host language. These libraries provide the hardware designer with basic building blocks for creating circuit descriptions. In the case of Lava, the host language (the glue to put the building blocks together) is the modern functional programming language Haskell [2].

In the talk, we argue in favour of the embedded language approach we have taken. The power of our approach comes from the fact that circuit descriptions in Lava are first-class objects in Haskell. This means that:

Circuit descriptions can be generated. We can write functions, that, given some parameters, generate a circuit. We can for example use lists and recursion to describe generic or size-independent circuits.

Circuit descriptions can be passed around as parameters. We can write functions that take circuits as parameters, and combine them to produce new circuits. We call these *connection patterns.*

Circuit descriptions can be analysed and transformed. We can define functions that inspect the structure of a circuit and generate a result. An example of this is a longest combinational path analysis. The result of such a function can also be another circuit, allowing us to define circuit transformations. Examples of this are retiming transformations.

Apart from the usual advantage that higher-order functions bring in programming, namely high-level code reuse, we get one more advantage from them in Lava: We can also give our connection patterns a layout semantics. For example, serial composition, written as `->-`, takes two circuits, and produces one circuit which feeds its input to the first circuit and the corresponding output to the second. In addition, the layout semantics says that the first circuit should

be laid out to the left of the second. Using a small amount of layout-aware connection patterns, and recursion, we are able to concisely describe very complex circuit layouts, such as butterfly networks [1].

Circuit descriptions can be processed in several different ways. We can simulate circuits, by running the Haskell functions corresponding to circuits on real inputs. We can also symbolically evaluate circuit descriptions, by instantiating parameters and providing symbolic inputs, to produce a low-level description of the circuit in another language. Thus, we can generate structural VHDL or EDIF, giving a route to implementation on a Field Programmable Gate Array (FPGA), as well as access to a variety of standard circuit analysis tools. Furthermore, we have equipped the Lava system with a property description language, with which we can describe properties about the circuits. Again, we use symbolic evaluation to generate logical descriptions of the circuits, which we can feed to a wide range of external model checking tools and automatic theorem provers. The output of these tools can be read back in, providing a way of scripting verification strategies from within the Lava system.

The work on Lava started several years ago, and has gone through a number of major revisions, but now we feel we have come to a point where a final design of the system can be presented. We have built a complete system in which real circuits can be described, verified, and implemented; it allows elegant circuit descriptions, formal and informal reasoning about the circuits, and a route to fast implementations on FPGAs. Future enhancements to the system are likely to be in the form of extensions to the embedded language, which will in turn demand new analysis methods. Lava has been used at Xilinx to design a wide range of circuits, such as several constant coefficient multiplier core generators for making high speed fully placed multipliers for use in graphics and signal processing applications, 2D convolvers for use in real-time 2D image correlation, sorting cores for calculating median values in image processing circuits, and distributed arithmetic implementations of trigonometric functions like sine, cosine etc. as well as filter functions.

To illustrate how designing in Lava can be very different from designing in a conventional hardware description language, we present the example of a constant coefficient multiplier (KCM) core targeting to Xilinx's Virtex FPGAs. By *core* we mean a generic (or parameterisable) circuit that efficiently performs high speed multiplications with predictable timing and regular floorplan. Such circuits are difficult to describe in a language like VHDL because it lacks several important features:

- In VHDL, circuits cannot be passed as parameters and returned as results;VHDL is not *higher order*. In Lava, the fact that the language is higher order allows us to construct a constant coefficient multiplier core by *computing* a circuit that depends on the constant coefficient.
- In VHDL, one cannot easily compose circuits together algorithmically. In Lava, we can produce a constant coefficient multiplier core with a variety of parameters (e.g. sizes of inputs, signed/unsigned data specification, registering options) by implementing an algorithm that determines which

specific architecture is best suited for a given instance. It is *possible* to write such descriptions in VHDL but many core developers resort to writing core generators in conventional programming languages like C and Java to generate VHDL or Verilog for a specific instance. Our approach has one language and description which is effective for describing both circuits and algorithms that compute circuits.

- In languages like VHDL, there is no standard and flexible mechanism for describing circuit layout. Specifying a good layout for a core allows performance to be tuned and predictable and also allows a core to have a regular floorplan, which makes it easier to use in a larger system. Lava provides a very flexible and powerful layout mechanism, intimately associated with the combinators that compose behaviour.

New hardware description languages like SystemC rectify some of the inflexibilites of conventional hardware description languages like VHDL and Verilog. For example, in SystemC, it is much easier to algorithmically compose structural circuit descriptions, since the full power of the C++ language is available. In SystemC, one can also exploit the template mechansim (which provides a more flexible form mechanism for specifying generic circuits than that in VHDL) and inheritence toproduce more general and extensible circuit descriptions. However, the features for supporting general descriptions in Lava (namely higher order functions and the polymorphic type system) are far stronger than their equivalents in SystemC and there are still many types of circuit descriptions which are elegantly rendered in Lava, but would be cumbersome in SystemC. Furthermore, there is still no standard mechnaism for describing circuit layout in SystemC.

The constant coefficient multiplier core

The constant coefficient multiplier core works by performing several 4-bit multiplications, the results of which are combined in a weighted adder tree to produce the final product. This is an efficient architecture for 4-input look-table based FPGAs since a 4-bit multiplication can be realized effectively with a look-up table. A block diagram of such a multiplier is shown in Figure 1. This KCM multiplies an 11-bit signed dynamic input $A = a_{10} \ldots a_0$ by an 11-bit constant coefficient K. The multiplication AK is performed by composing the results of several four-bit multiplications i.e. $AK = -2^8 a_{10\ldots8} K \ldots + 2^4 a_{7\ldots4} K + a_{3\ldots0} K$.

The rightmost table multiplies the lower four bits of the inputs by the K constant coefficient i.e. $X = a_{3\ldots0} K$ and the middle table computes $Y = a_{7\ldots4} K$ where both $a_{3\ldots0}$ and $a_{7\ldots4}$ are treated as unsigned numbers. The leftmost table calculates $Z = (a_{10}, a_{10}, a_9, a_8) K$ where $(a_{10}, a_{10}, a_9, a_8)$ is treated as a sign-extended bit-vector. To calculate the final results, the partial products Y and Z have to be appropriately weighted before addition. A weighted adder tree is used to compose the partial products and the size of the intermediate adders is reduced by reading off directly the bottom four bits of the partial product

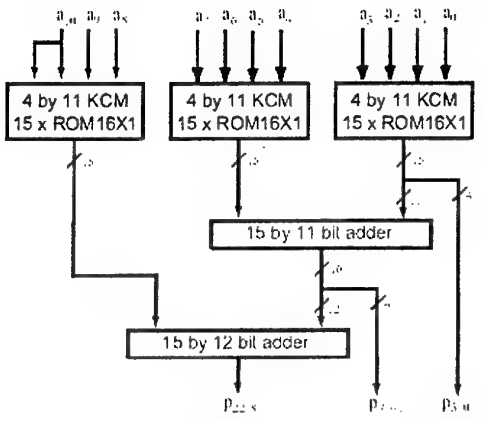


Figure 1: The structure of a constant coefficient multiplier

X and the result of adding the remaining bits of X to Y. For the pipelined version of this multiplier, we again exploit our ability to *calculate* circuits to place registers on the intermediate wires, making sure to balance the delays.

The adders in our KCM design are simply described by parameterised optimised adders from the Lava arithmetic library. But how should one describe the look-up tables for the 4-bit multiplications? The beauty of Lava is that we can write an generator circuit that takes as input an *arbitrary* Haskell function that maps any value that can be represented to a single bit and returns the corresponding look-up table circuit realisation of that function. Here, we have an example of mapping a circuit from one high level representation in our language to another more detailed representation (of the implementation) in the same language, with the algorithm for performing this mapping also being written in the same language.

The specific function that is used in the KCM implementation is called `rom16x` and has the following Haskell type:

```
rom16x :: Int -> [Int] -> (Bit, Bit, Bit, Bit) -> Bit
```

The function takes a size argument specifying the size of the table, a list of numbers to be represented by the table and a four bit value that at run-time determines which value of the table is present at the output. A function that performs a 4-bit constant coefficient multiplication can be represented by such a circuit builder since it can be realised as a table look-up of integer values using a 4-bit index.

Using the `rom16x` higher order circuit builder, we can define an unsigned four bit constant coefficient multiplier core as:

```
unsignedFourBitKCM :: Int -> [Bit] -> [Bit]
unsignedFourBitKCM coeif addr
  = rom16x maxwidth multiplication_results padded_addr
    where
    padded_addr = padAddress addr
    nr_addrs = length addr
    multiplication_results
      = pad_width 0 16 [coeif * i | i <- [0..2^nr_addrs-1]]
     maxwidth
      = maximum
        (map unsignedBitsNeeded multiplication_results)
```

The calculation of the constant coefficients is represented *directly in Haskell* by the expression

```
[coeif * i | i <- [0..2^nr_addrs-1]]
```

This expression cycles `i` through every value that can be represented using the given number of address bits and computes that value multiplied by the constant coefficient. This illustrates how Lava provides a multi-level language. At the highest level, one can write expressions using the full power of the Haskell programming language. At the next level, one can write aribitrary Haskell expressions to transform such expressions (in this case by exhuastive simulation) into a domain specific representation (combinators and primitive circuit elements represented in the core Lava language). The remainder of the definition computes the details of word-lengths.

An unsigned KCM can be made by using several 4-bit unsigned KCMs as follows:

- The input bus is chopped into groups of 4-bits. This is achieved using the `chop` list function.
- Each of these 4-bits is multiplied by the constant coefficient using the unsigedFourBitKCM circuit. This can be easily done by the expression `hmaP (unsignedFourBitKCM coeif)` which places the 4-bit KCMs next to each other.
- The results of each of these partial products have to be suitably weighted to allow them to be combined. This is done by zipping the partial products bus with the list [20, 24, 28...]
- A weighted adder tree is used to sum the partial products. Combinational and pipelined KCMs will have different adder trees but we would like to abstract over this difference so that one generic unsigned KCM function needs to be written. This can be done by passing the required adder tree circuit as parameter.
- The output of the weighted adder tree circuit is a pair which has 20 as the first element and the final product as the second element. Since we only want the product we just use the `snd` function to project it out.

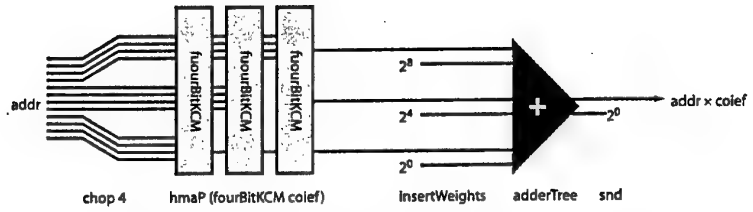


Figure 2: Architecture and Layout of an unsigned KCM

In Lava the steps described above can be represented by the following code:

```
unsignedKCM adderTree coeif
  = chop 4 >->
    hmaP (unsignedFourBitKCM coeif) >->
    insertWeights >->
    adderTree >->
    snd
```

This architecture is illustrated in Figure 2. The combinators used to compose the behaviour of the individual circuit components also compose the layout of the circuit. An example layout of the Lava generated KCM is shown in Figure 3. This specific KCM has been used in several real digital signal processing and image processing applications.

Conclusion

In conclusion, we observe that despite the emergence of new hardware description languages like SystemC there is still a role for declarative hardware description languages that provide multi-level descriptions within one framework. The ability to write arbitrary functions in a programming language and then pass these functions as arguments to circuit builders is a particularly powerful technique. This has been demonstrated successfully in the design and implementation of a constant coefficient multiplier core, and is typical of the way in which Lava is used in practice. We feel that we have only just begun to exploit the fact that we have a multi-level language. There are exciting possibilities for doing formal verification during circuit generation, instead of afterwards, for instance. In similar style, one can imagine analysing circuits for non-functional properties like timing, wire length or power consumption during the process of generating the final implementation. Having a multi-level language gives an extra phase on the way to a final circuit implementation, and so opens the way to performing various analyses, including formal verification, in new ways.

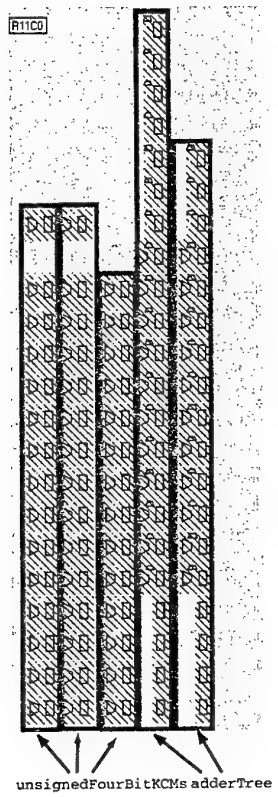


Figure 3: Layout of a KCM on a Xilinx Virtex FPGA

References

[1] Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *CHARME*. Springer, 2001.

[2] Simon Peyton Jones and John Hughes et al. Report on the programming language Haskell 98, a non-strict, purely functional language. Available from `http://haskell.org`, 1999.

Generic Operators for Circuit Synthesis and Optimisation.

Jean Vuillemin[1]

We highlight a technique for describing circuits within languages which support objects and generic operators – such as ***PAM-DC***, ***Jazz*** or ***Lava***. The ***source code*** is written in terms of generic operators applicable to all types of inputs. In the examples ***DCT*** and ***FHT*** chosen for illustration, they are simply the arithmetic operators: +, -, * and integer constants.

If we apply our ***source code*** to inputs of type *int* – the built-in integer type in the language – we obtain a software simulator for our target circuit. This simulator is efficient because it relies on efficient arbitrary precision arithmetic rather than bit level operations as normal circuit simulators would do. Having such an efficient software specification at our disposal is handy for: (a) verifying – in an exhaustive manner – if a particular choice of representation for the ***DCT*** coefficients meets or not the ***JPEG*** standard; (b) providing the physicists - who are the end-users of the detector in which ***FHT*** takes place - with an accurate and efficient software model, and let them adjust the critical parameters in the final algorithm.

If we apply the very same ***source code*** to inputs of type *net* – this type is built-in in ***Jazz*** – we obtain a ***bit-serial*** (base 2) circuit for implementing our algorithm: it has the same bit-wise behaviour as our previous simulator, modulo the I/O bit-order. Based on area and clock-speed, this would be the prime choice of representation for an ASIC implementation.

If we now apply our ***source code*** to inputs of type $serial_k$ – each arithmetic operation has to be constructed for this type – we derive ***digit-serial*** (base 2^k) circuits for our application. Each value of the integer parameter $k=1,2...$ corresponds to a different Area/Time trade-off. This tool proves critical in finding the sweet spots for various implementations in various technology. Points in the curve are been found optimal for: $k=2$ with X3K, $k=3$ with X4K and $k=4$ with ***CHESS***. Another input type is useful: the limiting case when $k=*$ is equal to the maximal binary length among operands in the application, and from which we automatically derive the ***bit-parallel*** circuit implementation. So we can systematically explore points in the Area/Time domain, from the smallest circuit $k=1$ to the largest $k=*$.

To further explore the ***hyper-serial*** part of this domain, we need two more technological ingredients: *block-memories* and *dynamic-instructions*. Both features are present in ***CHESS***; with a regular FPGA, dynamic-instructions can be realized by devoting some inputs to coding the operation. They let us implement the type $hyper_k$ from which we derive ***hyper-serial*** (base $2^{1/k}$) circuits: each gate in the circuit for $k=1/2$ realizes two gates from the ***bit-serial*** circuit $k=1$; each gets computed on alternate cycles, and one bit of memory is required per disappearing gate so as to store the intermediate value. This is done in a ***DCT*** connected to a single ported memory. Larger value of $1/k$ lead to smaller circuits performing more sequential processing through block-memories. The synthesized circuit in the limit $k=1/*$ is completely sequential, just as a software bit-level simulator.

[1] Ecole Normale Superieure, 45 rue d'Ulm, 75230 Paris Cedex 5, France.

Formal Verification of Microprocessors at AMD

April 6, 2002

Arthur Flatau
Matt Kaufmann (speaker)
David F. Reed
David Russinoff
Eric Smith
Rob Sumners

Advanced Micro Devices, Inc.

⌊ Today at AMD, page 1 ⌋

- Traditional simulation-based methods:
  - Block-level "whackers"
  - Full-chip directed tests written by hand
  - Full-chip test programs written by pseudo-random test generators
  - Various "checkers" monitoring simulation for potential bugs
- Boolean equivalence checking for comparing RTL (Register-Transfer Logic) models with custom gate-level models
  - Synthesis alone does not meet all our needs

⌊ Today at AMD, page 2 ⌋

- Formal verification using the ACL2 theorem-proving system

  – Proofs of correctness of RTL floating-point modules

    ∘ Specifically, proofs are done on the output from translation tools applied to the RTL.

  – Proofs of correctness of higher-level algorithms implemented in RTL

  – Ongoing improvement of ACL2 itself and libraries of lemmas used to "program" ACL2

This talk focuses on theorem proving and the consideration of more automatic formal methods.

[ACL2]

- ACL2 [1] is "A Computational Logic for Applicative Common Lisp"
(descendant of Boyer-Moore theorem prover)

- Authors: Matt Kaufmann and J Moore

- Interactive prover with induction, conditional rewriting, and decision procedures (arithmetic, equality, Boolean logic)

  – "Programmed" with theorems proved by the user, usually stored as rewrite rules.

- Publicly available at:
`http://www.cs.utexas.edu/users/moore/acl2`

  – Includes numerous papers and proof scripts, and links to ongoing work

- *[Plug]* ACL2 Workshop immediately follows this DCC Workshop.

⌊ Some AMD Formal Verification History ⌋

We have emphasized automated theorem proving.

- 1995–96: Division and square root algorithms for AMD-K5 microcode[3, 5]

- 1997–present: Proofs of floating-point algorithms and actual RTL that use ACL2 on the AMD Athlon^{TM} processor and its derivatives [6, 7, 8]

  – We have a translator from our proprietary RTL to ACL2 [7] that enables RTL proofs.

- 2001: Completed some protocol-level proofs

⌊ Floating-point Verification, page 1 ⌋

A natural target for theorem provers [10, 4]

- Concise formal specifications relating outputs to inputs

- The RTL is *relatively* tractable.

  – While the size of an FPU may be substantial, the logic tends to decompose by operation.

  – The interfaces with other modules are smaller and simpler.

- Complexity of floating-point designs causes problems for other verification approaches.

  – Testing alone may be inadequate.

  – Decision procedures used in formal verification traditionally have capacity limitations, for example for multiplication and shifting.

[Floating-point Verification, page 2]

We have addressed the verification of RTL models with increasing levels of complexity.

- Started with simple pipeline-based designs

- Conditional pipelines [2] allowed more complicated signal dependencies and the sharing of hardware among operations of different latencies.

- Current work involves RTL with feedback (especially state machines, which are used in the implementation of iterative algorithms).

⌊ Floating-point Verification, page 3 ⌋

Various tools besides ACL2 are involved in this verification effort.

- "Translator" (written in flex/bison/C++ and ACL2) takes RTL as input and generates forms in a Lisp-like target language for specifying state machine transitions.

  – We have also written high-level specs directly in this target language.

- "Compiler" (written in ACL2) analyzes signal dependencies and pipeline structures and produces ACL2 definitions.

⌊ Floating-point Verification, page 4 ⌋

- Tools (written in ACL2) automate repetitive tasks by generating lemmas automatically from the RTL:

  – Lemmas about bit-vector widths

  – Lemmas used in reasoning about conditional pipelines [2]

  – Lemmas connecting different models (combinational and executable)

- ACL2 library of general reusable lemmas [9] has been designed to simplify terms built from RTL operations, in many cases automatically.

  – Development continues on the RTL library, with users inside/outside of AMD [10].

⌊ Protocol-level Verification, page 1 ⌋

Formal verification of non-floating-point RTL can be considerably more difficult.

- Unclear and incomplete (or nonexistent) specs
- Decomposition of verification task is far more difficult.
  - Sufficient invariants often involve every state variable, and significant and complex environment assumptions are required.
- Experimental formal analysis of a bus interface unit (many thousands of lines of RTL)
  - instrumental in resolving a subtle liveness issue
  - limited practical value
- Higher-level proof attempt on cache correctness
  - Partially completed, but appeared to have limited payoff relative to the effort involved

⌊ Protocol-level Verification, page 2 ⌋

We completed proofs when the effort seemed justified.

- Proof of a write-ordering property with respect to a fairly sophisticated mechanism

  – Proof performed at algorithm level. Abstracted numerous uninteresting details. Formal analysis more effectively focused at subtle cases.

  – Informal statement: If processor P1 performs write Wr(addr1) followed by write Wr(addr2), and processor P2 performs reads at address addr2 and then addr1, then if the read at addr2 gets the new value, so does the write at addr1.

- Proof of progress for a routing module

  – The proof was performed on a model which generalized the RTL (i.e., the RTL was functionally equivalent to an instance of the model). The model was defined with recursive functions and data structures, which provided a much more expressive "language" for defining invariants, refinement maps, etc.

⌊ Attempts at Model Checking ⌋

We have begun looking at model checking and symbolic simulation, but initial results are lackluster.

- Our designs are in a proprietary language, which is not an input language for existing Model Checkers.
  - Translator output is often difficult for a human to read.
- Attempts at using symbolic simulation were ineffective due to incompleteness of search.
  - In order to expose bugs, we need to simulate for hundreds of cycles and simulation becomes inefficient much sooner than this.
- Attempts at using Bounded Model Checking have been more effective, but the property definition complexity is considerably higher.
  - Must explicitly define strengthened invariants.
  - Multiple modules expose expressiveness and capacity issues.

⌊ Problems ⌋

- Modules are large (many thousands of lines).

- RTL is not written in a standard language.

- It takes effort to develop meaningful specifications, which are not always readily supplied by the RTL developers.

- Is formal verification cost-effective?

  – RTL writers have told us that any value added would appear to be in verification involving interfaces among multiple large modules.

  – Time to write specs is a real issue, but has some support among the RTL designers.

  – We developed a simple checker (written in ACL2) for some sorts of typos that have been seen during pre-silicon RTL.

- Capacity, Capacity, Capacity...

References

[1] Kaufmann, M, Manolios, P, Moore, J S. *Computer-Aided Reasoning: An Approach.* Kluwer Academic Press, 2000.

[2] Kaufmann, M. and Russinoff, D., "Verification of Pipeline Circuits," in Proceedings of ACL2 Workshop 2000. Available at URL `http://www.cs.utexas.edu/users/moore/acl2/-workshop-2000/final/russinoff-kaufmann/paper.pdf`.

[3] Moore, J, Lynch, T., and Kaufmann, M., "A Mechanically Checked Proof of the Correctness of the Kernel of the $AMD5_K86$ Floating Point Division Algorithm", *IEEE Transactions on Computers*, 47:9, September, 1998.

[4] O'Leary, J., Zhao, X., Gerth, R., and Seger, C-J.H.,"Formally Verifying IEEE Compliance of Floating-Point Hardware", *Intel Technology Journal*, 1999.

[5] Russinoff, D., "A Mechanically Checked Proof of IEEE Compliance of the AMD-K5 Floating Point Square Root Microcode", *Formal Methods in System Design* 14:1, January 1999. See URL `http://www.onr.com/user/russ/david/fsqrt.html`.

[6] Russinoff, D., "A Mechanically Checked Proof of IEEE Compliance of the AMD-K7 Floating Point Multiplication, Division, and Square Root Algorithms", *Journal of Computation and Mathematics* 1, London Math. Society, December 1998. See URL `http://www.onr.com/user/russ/david/-k7-div-sqrt.html`.

[7] Russinoff, D. and Flatau, A., "RTL Verification: A Floating-Point Multiplier", in Kaufmann, M., Manolios, P., and Moore, J, eds., *Computer-Aided Reasoning: ACL2 Case Studies*, Kluwer Academic Press, 2000. See URL `http://www.onr.com/user/russ/david/acl2.html`.

[8] Russinoff, D., "A case study in formal verification of register-transfer logic with ACL2: The floating point adder of the AMD Athlon^{TM} Processor, in Hunt, Warren A. Jr. and Johnson, Steven D., eds., *FMCAD 2000, Proceedings of Third International Conference on Formal Methods in Computer-Aided Design*, Springer, November, 2000.

[9] Russinoff, D. and Smith, E., "An ACL2 Library of Floating-Point Arithmetic", to appear (earlier version at URL `http://www.cs.utexas.edu/users/moore/-publications/others/fp-README.html`).

[10] Sawada, J., "Formal Verification of Divide and Square Root Algorithms Using Series Calculation", to appear in Proceedings of ACL2 Workshop 2002.

Talk Abstract
Verifying a Commercial Microprocessor Design at the RTL Level

Ken McMillan
Cadence Berkeley Labs

February 28, 2002

Considerable progress has been made in the past few years in using formal methods to verify models of processors at the microarchitecture level. Successful verification of such models has been achieved using, for example, Burch-Dill style proofs based on efficient decision procedures or general purpose theorem provers, as well as proofs based on model checking and abstraction. However, these models, roughly at the level detail presented in computer architecture textbooks, are still far removed in complexity from the "RTL" level models used in the design of commercial microprocessors.

The complexity of commercial RTL-level designs stems from a number of factors, including low-level optimizations introduced by designers, the somewhat baroque nature of the instruction sets of most commercial processors, and the requirements of logic synthesis and simulation tools. The task of specification is further hampered by the fact that processor designs generally do not implement their instruction set architectures faithfully in all cases. Thus, correctness must be specified relative to certain restrictions on the instruction stream, which may not be precisely documented, and must be inferred from the design itself.

Using a commercial microprocessor design as an illustration, we will consider some approaches to specification and verification that make it possible to verify RTL-level processor designs against instruction set architecture models. We will cover a number of issues that arise in RTL-level verification that may not be encountered in more abstract models, including:

- Specification strategies that allow a greater degree of localization of the verification problem, to cope with the large size of the model.
- Tradoffs between verification at the bit level and at the word level (i.e., using uninterpreted functions).
- Tradeoffs between a direct refinement approach and the use of an intermediate model.
- Strategies for dealing with low-level asymmetries, such as variable-length instruction encodings, variable granularity of data transfer (bytes, words, cache lines), bit level manipulation of addresses (in cache controllers, for example), and so on.
- Specification approaches for dealing with incoherent caches and other implementation anomalies.

Although theses issues will be addressed in the context of a proof of correctness by compositional model checking, we will also consider the implications for proofs using the Burch-Dill approach.

Verifying a Commercial Microprocessor Design at the RTL level

Ken McMillan
Cadence Berkeley Labs
mcmillan@cadence.com

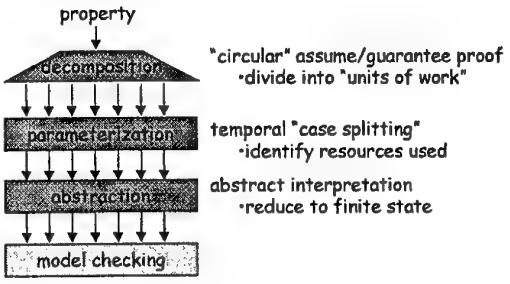
We will consider some of the problems involved in verifying the actual RTL code of a commercial processor design, as opposed to an architectural model.

This is a work in progress...

Outline

- Methodology
- The PicoJava design
- Verification Strategy
- Problems

Proof Methodology



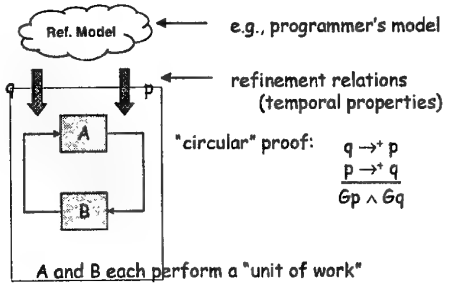
"Circular" assume/guarantee

- Let $p \rightarrow^+ q$ stand for
  "if p up to time t-1, then q at t"
- Equivalent in LTL of
  $\neg(p\ U\ \neg q)$
- Now we can reason as follows:

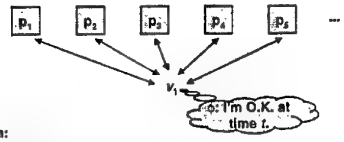
$$\frac{q \rightarrow^+ p \qquad p \rightarrow^+ q}{Gp \wedge Gq}$$

That is, if neither p nor q is the first to be false, then both are always true.

Using a reference model



Temporal case splitting



Idea:
parameterize on most recent writer w at time t.

$$\frac{\forall i: G((w{=}i) \Rightarrow \phi)}{G\phi}$$

Abstract interpretation

- Problem: variables range over unbounded set U
- Solution: reduce U to finite set $\hat{U}$ by a parameterized abstraction, e.g.,

  $\hat{U} = \{\{i\}, U\backslash i\}$

  where U\i represents all the values in U except i.
- Need a sound abstract interpretation, such that:

  if ϕ is valid in the abstraction, then, *for all parameter valuations*, ϕ is valid in the original.

Data type abstractions in SMV

- Examples:
  - Equality

$\hat{=}$	{i}	U\i
{i}	1	0
U\i	0	⊥

represents "no information"

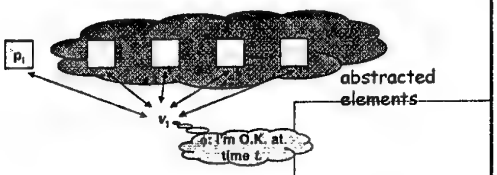
  - Function symbol application

x	{i}	U\i
$\hat{f}(x)$	$\hat{f}(i)$	⊥

Unbounded array reduced to one fixed element!

Note: truth value under abstraction may be ⊥...

Applying abstraction



Must verify by model checking:

$\phi \rightarrow^{+} ((w{=}i) \Rightarrow \phi)$

i.e, if p_i is the most recent to modify v_1, then v_1 is correct.

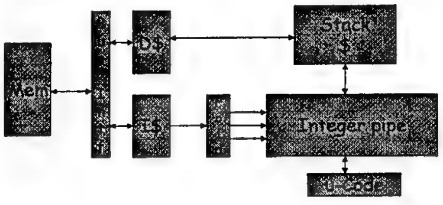
Review

- By a sequence of three steps:
  - "circular" assume/guarantee reasoning
    (restricts to one "unit of work")
  - case splitting (adding parameters)
    (identifies resources used in that unit of work)
  - abstraction interpretation
    (abstracts away everything else)

...we reduce the verification of an unbounded system of processes to a finite state problem.

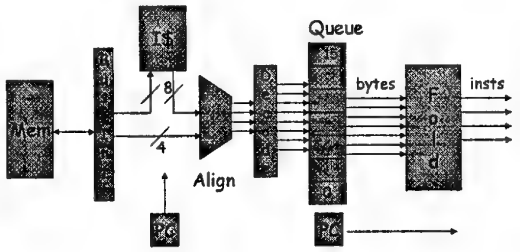
PicoJava

- Stack machine architecture
- Implements Java bytecode interpreter in hardware



Instruction path

- We will concentrate on I$ and Fold units.



Specification strategy

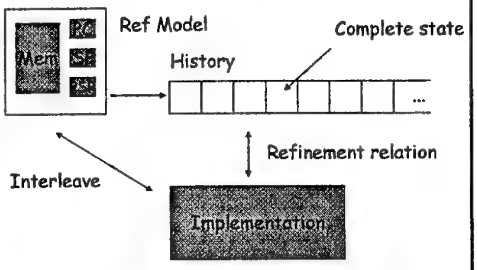
- Since implementation is very large and complex, we need a specification strategy that allows a fine-grain decomposition of the proof.
- Topics:
  - Reference Model
  - Histories
  - Tags and Refinement Relations
  - Dealing with Exceptions

Reference Model

- Programmer's view of Java machine (ISA)
  - contains only programmer visible state

Relating Impl to Ref Model

- Specify Impl w.r.t. reference model history



Correctness criterion

- Correctness is defined as follows:
  - There exists some interleaving of Impl and Ref, such that the given relation holds between Impl and history.
- Must choose a witness interleaving
  - Any interleaving that ensures reference model "stays ahead of" the implementation.

We use this approach because one step of implementation may correspond to many steps of reference model.

Multiple histories

- Instructions are a variable number of bytes
- Some parts of Impl deal with bytes, some with instructions.
- Keep two histories:
  - Byte level history (stream of instruction bytes)
  - Inst level history (stream of instructions)

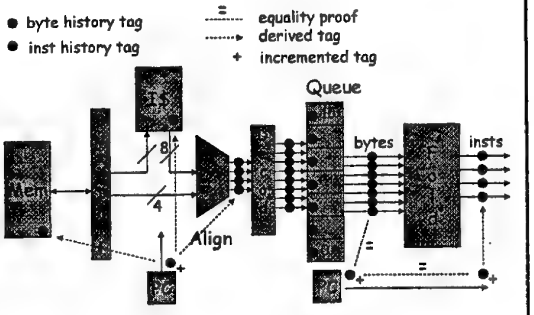
We could also record history at coarser granularity if needed...

Tags and refinement relations

- Tags are auxiliary state information
- Tags are pointers into a history (byte or inst)
- Tags flow with data
- Refinement relations
  - Are temporal specifications of data correctness
  - Use tags to locate correct value of data in history

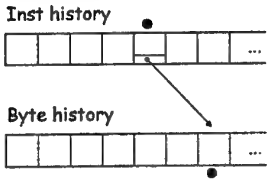
Note, we sometimes have to prove equality of tags to show correct data flow

Tags for instruction path



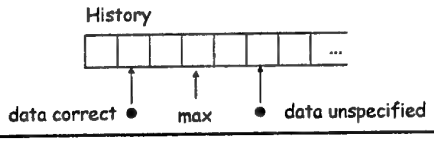
Alignment between histories

- Comparing tags into byte and inst histories
  - record byte history position of each inst



Dealing with Exceptions

- Exceptions (e.g., branch mispredictions)
  - pipeline may be executing incorrect instructions
  - incorrect instructions must be flushed
- Specification strategy
  - Define tag "max"
    - latest instruction correctly fetched
  - Data with tag after "max" is unspecified



Summary of approach

- Strategy
  - Reference model/ Histories/ Tags
- Localization of verification
  - Model checking can be localized to very small scale.
  - State explosion is not a problem.

Problems

Accidents happen to words

- Verification depends strongly on abstraction of data types.
  - Use uninterpreted types and functions.
  - 32-bit word might be abstracted to:
    { a, b, ~ }
  where a and b are parameters of a property.
- Problem:
  - In RTL descriptions, words are often arbitrarily broken into bits and reassembled.

Example accident

- 8-bit register implemented in cells:

```
module reg8(clk,inp,out);
  input clk, inp[7:0];
  output out[7:0];
  reg1 cell0(clk,inp[0],out[0]);
  ...
  reg1 cell7(clk,inp[7],out[7]);
endmodule
```

The state is actual held in bits.
How do we abstract the state?

Example Accident

- Verilog can't make 2-D arrays!

```
module foo(bits,...);
  input bits[63:0];
  byte0 = bits[7:0];
  ...
  byte7 = bits[63:56];
...
```

Instead of an array of bytes, we get 64 bits!

A pragmatic approach

- If possible, verify property at bit level
  - Words must not index large arrays
  - Can use "bit slicing"
- Else, use two-level approach
  - Make intermediate model at word level
  - Verify properties using abstractions
  - Verify intermediate model at bit level

This avoids re-modeling the entire design using uninterpreted types and functions.

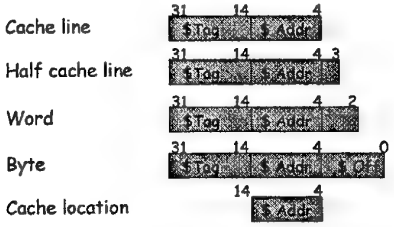
Bit-field abstractions

- Words are often divided into fields

- Typical abstraction
  - property has parameters t ($ Tag) and a ($ Addr)

But accidents happen...

- Adresses of many different bit lengths occur



Since types are not structured, how does a tool know how to divide and abstract these bit vectors?

Manual approach

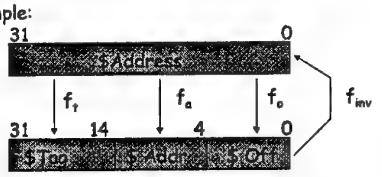
- Re-model using structured types
  - i.e., instead of a bit vector, use:

```
struct {
  tag : $TAG;
  addr : $ADDR;
  offset :  array 3..0 of boolean;
}
```

- Prove model correct at bit level
- Prove property using type-based abstractions
  - examples: cache contents correctness, aligner output, etc...

Mapping between representations

- Sometimes need to translate between representations with uninterpreted functions
  - example:



(Must manually instantiate injectiveness axiom)

What's needed?

- Ability to abstract any bit-field of a word
  - conceptually straightforward
- Some heuristic method of grouping bits together and assigning them types?
  - less obvious

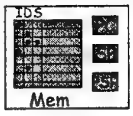
Essentially, we need to be able to reverse-engineer a bit-level design into a structured design.

Incoherence

- Few processors implement ISA precisely
  - makes writing a specification difficult
- Example: three incoherent caches in PicoJava
  - Instruction (I)
  - Data (D)
  - Stack (S)
- How to handle mismatch between ISA and Impl?

Solution (?)

- Mark every address as valid/invalid for I,D,S

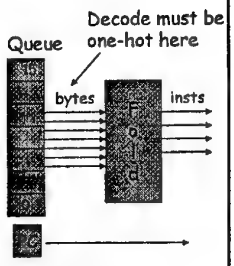


- Example:
  - I becomes valid when I$ line explicitly flushed
  - I becomes invalid when location written as data
- *Assume* program never reads invalid addresses

  Problem: Pipe delay means address is readable unknown number of clock cycles after flush instruction (???)

Accidental correctness

- Example:
  - decode not one-hot until first queue load (!)
  - but, in PSR, Fold unit not enabled at reset
  - one instruction required to enable Fold unit
  - hence one-hot when Fold unit enabled!



Note, local property (one-hotness) depends on far away logic (PSR, integer unit, etc...). This is not written anywhere because no one actually knows why circuit works!

Conclusions (?)

- Compositional verification of real processors at RTL level *is* possible.
  - Reference model/ Histories/ Tags
- Several aspects of typical RTL descriptions make it much more difficult:
  - Bit-level representation of words
  - Lack of structured data types
  - Accidental correctness

Design for formal verification could largely correct these problems.

An Introduction to Abstract Interpretation

Nicolas Halbwachs
Vérimag/CNRS, Grenoble

Abstract

Abstract interpretation has been under development for more that twenty years [CC77]. It can be viewed both as a unified theory of *approximation* in dynamic systems, and as a generic technique for analysing these systems. It has mainly been applied to program analysis (e.g., [CC92a, NNH99]).

In this short tutorial, we present the principles of abstract interpretation from the point of view of its application to verification. In this context, abstract interpretation has mainly been used [CGL94, GL93] to reduce infinite state systems to finite ones: the state space in *quotiented* into a finite number of classes, and the considered system is interpreted over these classes. Applying standard techniques to this abstract system allows one to compute, for instance, over-approximations of the reachable states of the system.

This finite quotienting of infinite state spaces uses only a part of the abstract interpretation technology. The general approximation technique can be applied to infinite state systems, and combines *abstraction* with *extrapolation*, which is a way of "guessing" the limit of a computation. [CC92b] shows that this general technique can be strictly more powerful, but its applications to verification (e.g., [HPR97]) are not widespread.

References

[CC77] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1–4):103–179, 1992. (Also, Research Report LIX/RR/92/08, Ecole Polytechnique).

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP'92*, Leuven (Belgium), January 1992. LNCS 631, Springer Verlag.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5), 1994.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[HPR97] N. Halbwachs, Y.E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997.

[NNH99] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis.* Springer-Verlag, 1999.

Modular analysis of a circuit description language by abstract interpretation: Application to the automatic extraction of circuit shapes.

Charles Hymans
LIX, École Polytechnique, 91128 Palaiseau, France.

February 1, 2002

Abstract

We design a static analysis that extracts the "shapes" of a circuit described in structural VHDL. It automatically computes a superset of all the possible connections between wires of the circuit.

This analysis is able to infer and describe connections between arrays of wires of arbitrary size in a precise way. Although precise, it is still efficient. First, we use a compact representation to summarize the possible connections of a design. Second, the analysis is modular: it can be run on any component, the result can then later be reused for analyzing a code that instantiates that particular component.

The result of our analysis helps programmers getting better insights about their code. It allows a quick understanding of the dependences inside a module and provides a way to estimate the impact of a piece of code.

1 Introduction

Today, circuits are specified using Hardware Description Languages. Modern HDL, such as VHDL or Verilog, allow different levels of description to coexist. The classical development scheme in VHDL is to first sketch the

```
A <= B or C;
```

Figure 1: Wire connection

```
entity example is
  generic(n : integer);
  port(A : bit_vector(1 to n), B : bit_vector(1 to 2*n));
end;
architecture rtl of example is
  for I in 1 to n
    B(I) <= not A(2*I);
  end;
end;
```

Figure 2: Component

functionality of a system at a very high level of abstraction and then to refine this description until it can be automatically synthesized. A structural VHDL description specifies how basic gates and components (circuits) are to be connected in order to produce the final circuit. For instance the very simple program in figure 1 gives rise to a circuit where the wire `A` is connected to both wires `B` and `C` through a or-gate. From a structural VHDL description, our goal is to automatically infer the structure of the interconnections between wires. That is, in the case of code 1, we want to report that `A` is connected to `B` and `C` and that no other connection exists. We forget the fact that this connection occurs through a or-gate. Thus we extract the "shapes" of circuits described in structural VHDL.

This task seems very easy in the case of such a simple example. One could use a graph whose nodes are variables names and edges represent the possible connections. But, now consider the module described in program 2. The module takes an integer `n` and two arrays, `A` and `B`, of wires . It connects element i of `B` to element $2*i$ of `A`. Thanks to parameter `n`, an infinite family of circuits is here described. And, there is no a priori bound on the sizes of both `A` and `B`. It is thus more difficult to represent the possible connections induced by program 2. One possible solution is to restrict ourselves to programs without any free variables. Then we can expand the body of each component as many times as it is instantiated and also unroll loops. Every array of

the resulting program is now of finite length. Distinguishing between every elements of the arrays, we can apply the same technique as before. This method leads to very precise results. However it has many drawbacks: we can not analyze pieces of code outside their instantiation context; we must handle very expensive data-structure; the size of the code has blown up. Maybe more importantly, the program transformation makes the analysis results difficult to report to the programmer. As an alternative, we can decide to represent a whole array by only one node in the shape graph. Unfortunately, this leads to very imprecise results. For instance, the shapes computed for example 2, would contain an edge from node `A` to node `B` that expresses the fact that any element of `A` may depend on any element of `B`.

These two solutions are clearly both unsatisfactory. In order to come up with an efficient but still precise analysis, we conceived a compact data-structure that accurately summarizes the possible connections of any part of a circuit. As before, it consists in a graph whose nodes are variable names and edges stand for connections. This graph is enriched with numerical constraints that further refine the set of circuits it represents. We label an edge between two arrays by the numerical relation that holds between the indexes of the elements that are connected. In case of code 2, the analysis leads to a graph with an edge from `B` to `A` that is labeled by the constraint $2 * l = r$. This means that the l^{th} element of `B` may be connected to the r^{th} element of `A`, as long as the relation $2 * l = r$ holds. This happens to be a very accurate description of the shapes of the connections. Our abstract data-structure is parameterized by the underlying numerical constraint domain. Different domains may be plugged in to tune the precision/efficiency ratio.

The results of our analysis help programmers understand code they have not written. It allows them to quickly grasp the dependences between the variables of a piece of code. This may be particularly helpful when re-engineering or reusing code written by others. In the case of critical hardware system, one wants to estimate the impact of a failure of a particular component in the whole design. This can also be done with the help of our analysis.

VHDL program slicing techniques as in [CFR+99] and [INIY96] provide the same kind of facility. However they make the very crude approximations we pointed out before. That is, individual elements in arrays are not distinguished. Slicing is applied to whole VHDL whereas we restrict ourselves to the small RTL subset of the language. We made this on purpose so as to simplify the presentation and be able to put more emphasis on the novelties

$$
\begin{array}{rcl}
\mathit{program} & ::= & \mathit{module}^* \\
\mathit{module} & ::= & \texttt{entity}\ \mathit{ident}\ \texttt{generic}\ \mathit{decl}^*\ \texttt{port}\ \mathit{decl}^* \\
 & & \texttt{architecture}\ \mathit{statement}^* \\
\mathit{decl} & ::= & \mathit{ident} : \mathit{type} \\
\mathit{type} & ::= & \texttt{integer} \mid \texttt{bit} \mid \texttt{bit\_vector} \\
\mathit{statement} & ::= & \mathit{assignment} \mid \mathit{block} \mid \mathit{for\_loop} \mid \mathit{instanciation} \\
\mathit{assignment} & ::= & \mathit{expr}\ \texttt{<=}\ \mathit{expr} \\
\mathit{block} & ::= & \texttt{block}\ \mathit{decl}^*\ \mathit{statement}^* \\
\mathit{for\_loop} & ::= & \texttt{for}\ \mathit{ident}\ \texttt{in}\ \mathit{expr}\ \texttt{to}\ \mathit{expr}\ \mathit{statement}^* \\
\mathit{instanciation} & ::= & \mathit{ident}\ \texttt{generic map}\ \mathit{ident}^*\ \texttt{port map}\ \mathit{ident}^* \\
\mathit{expr} & ::= & \texttt{1} \mid \texttt{TRUE} \mid \mathit{ident} \mid \mathit{ident}(\mathit{expr}) \\
 & \mid & \mathit{expr}\ \texttt{+}\ \mathit{expr} \mid \mathit{expr}\ \texttt{and}\ \mathit{expr} \mid \mathit{expr}\ \texttt{=}\ \mathit{expr} \mid \ldots
\end{array}
$$

Figure 3: Abstract syntax of RTL VHDL

of our approach. We nevertheless believe our shape analysis can be extended to handle all the constructs of VHDL.

2 Syntax

Figure 3 presents the abstract syntax of the VHDL subset we deal with. For concision reasons, this syntax is a lightened version of the heavily decorated official syntax of the IEEE standard [ANS88]. A program is a set of modules. Each module has an interface visible from the outside, the entity, and an internal definition, the architecture. The interface specifies the arguments passed during a component instantiation. The arguments of a module are of two kinds: ports connect a component to the rest of the circuit, and generics allow parameterization of the module. The architecture explicits the structure of the component by a set of statements. Statements are assignment, block, for loop and module instantiation. An assignment connects wires through gates. A block allows the introduction of local variables. Note that the for loop also automatically introduces a local variable: the index *ident* of the loop. Module instantiation is a way to reuse previously defined

$$
\begin{array}{rcl}
\textit{instruction} & ::= & \textit{command} \mid \mathsf{call}\ \textit{ident}\ (\textit{ident}/\textit{ident})^* \\
\textit{command} & ::= & \mathsf{create\_int}\ \textit{ident}\ \textit{expr} \mid \mathsf{del\_int}\ \textit{ident} \\
& \mid & \mathsf{assert}\ \textit{expr} \mid \mathsf{assign}\ \textit{ident}\ \textit{expr} \\
& \mid & \mathsf{create\_sig}\ \textit{ident} \mid \mathsf{del\_sig}\ \textit{ident} \\
& \mid & \mathsf{connect}\ \textit{expr}\ \textit{expr} \\
& \mid & \textit{command}; \textit{command}
\end{array}
$$

Figure 4: Control flow graph's instructions

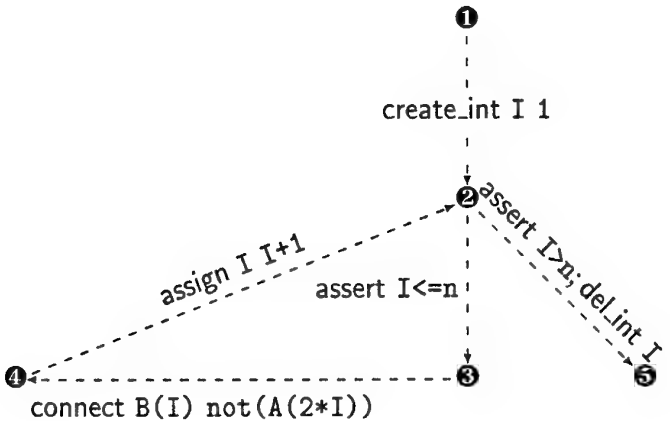


Figure 5: Control flow graph

components.

Without any loss of generality, we suppose that all variables have distinct names, so that it becomes unnecessary to rely on the lexical scope to differentiate them. Variables are typed: a variable is either an integer (*int*), a wire (*bit*) or an uni-dimensional array of wires (*bit_vector*). We call signals variables of type *bit* or *bit_vector*.

3 Operational semantics

We give the semantics of programs in two steps. First, we translate each module from the source code into a control flow graph. An edge in the control flow graph links two program points and is labeled by a simple instruction. The set of instructions stems from the syntax of figure 4. For example, the program of figure 2 is translated into the control flow graph of figure 5. Note that the language of instructions includes the composition of two commands. This means that one complex construction of our original language may be translated into the composition of simpler instructions. Expressing the constructions of the language as a composition of simple operators reduces the cost of designing a static analysis. The soundness proof is smaller, the many but often somewhat redundant constructions of the source language may be handled quickly. Also, the analysis can be applied to any other language (like RTL Verilog) whose semantics can be expressed by composing the basic operators. The trick, however, is to keep the set of nuclear instructions both concise and expressive enough. We denote by $l \xrightarrow{\mathsf{i}} l'$ the fact that there is an edge from l to l' which is labeled by the instruction i. Also, for every module g, $entry(g)$ and $exit(g)$ respectively denote the entry and exit program points in the graph of g.

Second, we define the semantics of programs in an operational fashion [Plo81]. Programs are run on an abstract machine whose possible execution steps are described by a transition relation. The states of the machine are tuples of the form (l, ρ, C, S). A label l indicates the current point of execution in the program. An environment ρ maps variables to values. A value may either be an integer, a wire or an uni-dimensional array of wires. The execution of a RTL VHDL program leads to the construction of a circuit. The circuit C is a collection of tuples (w, T), where w is a wire and T is some term built up from logical gates and wires. Lastly the stack S is needed to mimic the calling mechanism of modules.

The non-inductive rules in figure 6 explains the possible steps for the abstract machine. When a module is called, a new environment, which maps the formals to the values of the actuals, is created. The previous environment is pushed onto the stack. Returning from a module amounts to restoring the previous environment by popping the stack. For all other instructions the environment and the circuit are modified according to the semantics of the commands.

$$\frac{l \xrightarrow{\mathsf{c}} k \wedge \mathsf{c} = \mathsf{call}\ g\ f_i/a_i \wedge l' = entry(g) \quad \rho' = \mathsf{create\_env}_{f_i/a_i}(\rho) \wedge S' = S.(l,\rho)}{(l,\rho,C,S) \to (l',\rho',C,S')} \text{ call}$$

$$\frac{S'.(k,\rho') = S \wedge k \xrightarrow{\mathsf{call}} \cdots l'}{(l,\rho,C,S) \to (l',\rho',C,S')} \text{ return}$$

$$\frac{l \xrightarrow{\mathsf{c}} l' \wedge \mathsf{c} \in command \quad (\rho',C') \in [\![\mathsf{c}]\!](\rho,C)}{(l,\rho,C,S) \to (l',\rho',C',S)} \text{ execute}$$

Figure 6: Operational semantics

The semantics of commands is specified by rules like:

$$(\rho',C') \in [\![\mathsf{assert}\ bexpr]\!](\rho,C) \iff \begin{cases} [\![bexpr]\!]\rho = true \\ \rho' = \rho \wedge C' = C \end{cases}$$

$$(\rho',C') \in [\![\mathsf{connect}\ lexpr\ rexpr]\!](\rho,C) \iff \rho' = \rho \wedge C' = C \cup \{(w,T)\}$$

$$\text{where} \begin{cases} w = [\![lexpr]\!]\rho \\ T = [\![rexpr]\!]\rho \end{cases}$$

This means that assert *bexpr* allows execution to continue if the current state makes the boolean expression *bexpr* evaluate to true. Furthermore, connect *target expr* creates a new connection in the circuit between the wire denoted by *target* and the term denoted by *expr*. For the other commands let us just state informally that commands create_int, del_int and assign manipulate integer variables: they respectively create a new integer variable, destroy it and assign to it a new value. Commands create_sig and del_sig deal with signals. create_sig creates a new wire or an array of wires distinct from all previously existing wires. del_sig eliminates a signal variable from the environment.

We denote by X_0 the set of initial states of the program. It contains all the states (l,ρ,C,S), where l is the entry point of the main module, ρ maps free variables of the module to any possible value of their type, the circuit C and the stack S are both empty.

An execution trace of a program is a sequence of states $s_0 \dots s_n$ such that s_0 is an initial state and such that any two consecutive states $s_i s_{i+1}$ in the trace are linked by the transition relation. Trace semantics groups all possible (partial and complete) execution traces of a program, and it can be expressed as the least fixpoint of the following continuous function:

$$T(X) = X_0 \cup \{s_0 \dots s_n s_{n+1} \mid s_0 \dots s_n \in X, s_n \rightarrow s_{n+1}\}$$

Trace semantics is our concrete model of program execution. To design our shape analysis algorithm, we follow the methodology of abstract interpretation. We come up with an abstract domain that allows us to compute the property of interest (namely the shapes of the circuit built by a RTL VHDL program). Abstract and concrete semantics are related through a Galois connection. The fact that Galois connections may be composed allows us to reduce the complexity of the approach: we do this in several successive steps. The first step being a modular semantics.

4 Modular semantics

It is an awkward task to design a static analysis that is precise and at the same time scales up. Algorithms that scale up usually give rough information whereas precise algorithms tend to be very costly. One way to avoid this dilemma is to design modular analyses. Such analyses compute results, one module at a time, from the source of the module and from previous results only.

The cornerstone of such an approach is a semantics which collects the segments of execution that occur inside each module separately. Segmented semantics can be given as the least fixpoint of the following continuous operator:

$$\begin{aligned}
S(X)(g) \;=\; & \{s_0 \mid l_0 = entry(g) \wedge s_0 \in X_0\} && (1)\\
\cup\; & \{s_0 \mid t_0 \dots t_m \in X(f) \wedge l_m \xrightarrow{\mathsf{call}\ g} k \wedge t_m \rightarrow s_0\} && (2)\\
\cup\; & \{s_0 \dots s_n s_{n+1} \mid s_0 \dots s_n \in X(g) \wedge l_n \xrightarrow{\mathsf{call}\ h} l_{n+1} \wedge s_n \rightarrow t_0 \wedge \\
& \qquad t_0 \dots t_m \in X(h) \wedge t_m \rightarrow s_{n+1}\} && (3)\\
\cup\; & \{s_0 \dots s_n s_{n+1} \mid s_0 \dots s_n \in X(g) \wedge s_n \rightarrow s_{n+1}\} && (4)
\end{aligned}$$

where every l_i denote the label of the state with same index i

The traces corresponding to a given module g are built in the following way:

- if g is the main module, then the initial states are added (1)
- if the traces of a module f leads to a call to module g, then the first state after this call is added (2)
- if one of the execution traces ends onto a module call to h, that the execution within h can reach the end of the module and return, then we extend the trace by the state after returning from the call (3)
- as long as the execution remains within the same module g, then the trace is simply extended (4)

We show the connection between trace and segmented semantics. In fact, we have defined a complete join morphism[1] α_{chop} and we have shown that segmented semantics is a sound and complete approximation of trace semantics through the abstraction α_{chop}. In other words we have the soundness:

$$\alpha_{chop}(\text{lfp } T) \subseteq \text{lfp } S$$

and the completeness:

$$\text{lfp } S \subseteq \alpha_{chop}(\text{lfp } T)$$

Unfortunately, because of equation (2), the semantics of a module g can not yet be computed only from its source code and the results for all the modules it calls. To remedy this, we make a crude approximation and do not restrict the contexts in which g can be called. We can thus replace (2) by the simpler:

$$\{s_0 \mid l_0 = entry(g)\}$$

For the purpose of shape analysis, we wish to collect in each program point the parts of the circuit that have been created since the beginning of the execution of the module. Hence we further abstract our semantics thanks to the following complete join morphism:

$$\alpha(X)(g)(l_n) = \{(\rho_n, C_n \setminus C_0) \mid (l_0, \rho_0, C_0, S_0) \ldots (l_n, \rho_n, C_n, S_n) \in X(g)\}$$

The resulting semantics is obtained as the least fixpoint of the continuous operator M of figure 7. This operator is the basis on which we develop the shape analysis algorithm.

[1] A complete join morphism uniquely determines a Galois connection, see [CC92]

$$
\begin{aligned}
\mathsf{entry}(g) &= \{(\rho, C) \mid l = entry(g)\} \\
\mathsf{plug}_{f_i/a_i}(X)(Y) &= \{(\rho, C \cup C') \mid (\rho, C) \in X \wedge (\mathsf{create\_env}_{f_i/a_i}(\rho), C') \in Y\} \\
M(X)(g)(l) &= \mathsf{entry}(g) \\
&\cup \bigcup\{\mathsf{plug}_{f_i/a_i}(X(g)(l'))(X(h)(exit(h)) \mid l' \xrightarrow{\mathsf{call}\ h\ f_i/a_i} l\} \\
&\cup \bigcup\{[\![\mathsf{c}]\!](X(g)(l')) \mid l' \xrightarrow{\mathsf{c}} l\} \\
A &= \mathrm{lfp}\ M
\end{aligned}
$$

Figure 7: Operator M

5 Shape analysis

The operator M manipulates infinite sets of pairs environment/circuit. Obviously, this makes the least fixpoint of M not computable. We design an abstract domain that represents possible connections in the circuit, i.e. its shapes. We then derive the abstract counterpart $\mathsf{init}^\sharp$, $\mathsf{plug}^\sharp$ and $[\![\mathsf{c}]\!]^\sharp$ for each of the operations that appear in the definition of M. An abstract operator $M^\sharp$ is defined by simply replacing init, plug and $[\![\mathsf{c}]\!]$ by their abstract correspondent in the equation of M. The local soundness of the abstract operations induces the soundness of lfp $M^\sharp$ with respect to lfp M. Our algorithm then consists in computing the least fixpoint of $M^\sharp$. Thanks to Tarski's fixpoint theorem, this can be done by computing $M^\sharp(\bot)$, $M^\sharp(M^\sharp(\bot))$, $M^\sharp(M^\sharp(M^\sharp(\bot)))$, ... until stabilization. Lastly, the fact that our abstract domain checks the ascending chain condition[2] insures the termination of our analysis. Practically, more efficient fixpoint algorithms can be used, see [HDT87, Bou93].

As mentioned in the introduction, a dependence graph D is our abstract representation for shapes of circuit. The nodes of D are signal variables[3]. An edge between two nodes `A` and `B` indicates that there may be a connection from `A` to `B`. In the case where either `A` or `B` is an array, then the edge is labeled by a symbolic representation of the relationship that holds between the indexes of the elements that are connected. Many symbolic representation for constraints between integer variables have been proposed within the framework of abstract interpretation. Some are [Kar76, Min01, CH78, Gra97].

[2] We can also use widening operators

[3] Recall that a signal is either a wire or an array of wires

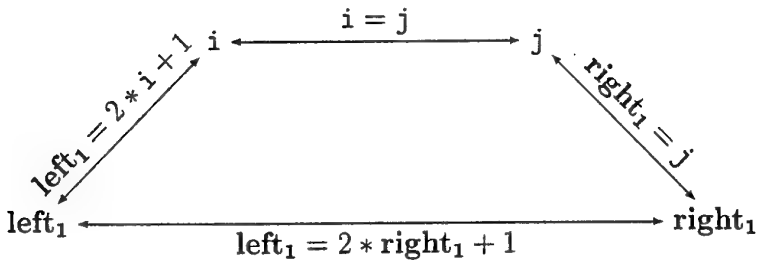


Figure 8: Pivot role of the environment

In order to be able to infer a dependence graph without too much loss of information, we also need to keep track of the possible values of the integer variables in the program. Consider the following assignment statement:

```
A(2*i+1) <= B(j);
```

Without knowing anything about the possible values of `i` and `j`, we can not do better but infer that some element of `A` is connected to some element of `B`. This is different in the case where we know that `i` = `j`.

So, every abstract operation transforms pairs (E, V) of a numerical domain and a dependence graph. We refer the reader to [Hym01] for their complete definitions. Now, suppose we use the numerical domain of linear equality [Kar76]. Let us sketch the action of $[\![$`connect A(2*i+1) B(j)`$]\!]^\sharp$ on an environment where `i` = `j` and an empty dependence graph. This will keep the environment unchanged and add an edge from `A` to `B` in the graph. What constraint labels this edge ? Let the variables $\mathbf{left_1}$ and $\mathbf{right_1}$ denote the index of `A` and `B`. From the expressions, we infer that $\mathbf{left_1} = 2 * \mathtt{i} + 1$ and $\mathbf{right_1} = \mathtt{j}$. The environment tells us that `i` = `j`. We put all these constraint together, project onto $\mathbf{left_1}$ and $\mathbf{right_1}$. This results into the constraint $\mathbf{left_1} = 2 * \mathbf{right_1} + 1$. Figure 8 pictures the pivot role that the environment has played in this process.

6 Implementation

We implemented the analysis in ML. It is long of approximately 3000 lines of code. The VHDL code is parsed. From the syntax tree, we build the

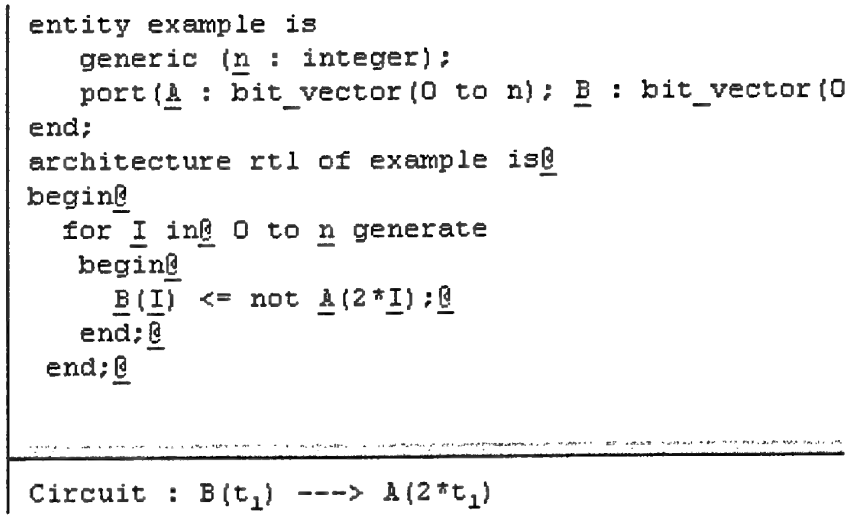


Figure 9: Snapshot

abstract domain and the abstract equations. Once this is done, a fixpoint computation algorithm is run. It traverses the control flow graph of the program in reverse post order and fires the abstract operations in turn. In the end, the result is output as html files. A snapshot from the analyzer is presented in figure 9.

We were disappointed to find out that the analysis does not radically improve results of simpler technique like in [CFR+99]. The reason for this, is that most of the time, in realistic RTL VHDL programs, arrays are connected in very simple ways: so that the only extra information we are able to gather is the fact that the indexes of two connected arrays coincide.

7 Conclusion

We presented the design of an analysis that extracts the possible connections of a circuit described by a RTL VHDL code. We expressed the semantics of the constructions of the language thanks to the composition of a small number of operators. This makes our analysis applicable to any language whose semantics can also be expressed with these operators (e.g RTL Verilog). The analysis may be computed one module at a time. We believe this modularity allows it to scale up. Unfortunately, the practical result turned out to

be somewhat disappointing: considering realistic RTL VHDL programs we think our technique may be an overshot. However, our theoretical results, in particular the modular semantics and the domain of dependence graphs enriched with numerical constraints, are general and we hope they may be reused in other works.

We believe that, as future work, it would be interesting to incorporate informations about the time dependences: i.e to collect properties of the form such output depends on such input with a delay of 3 cycles (or 10 ns).

Acknowledgement

We wish to thank Radhia Cousot, Xavier Rival, Francesco Logozzo and Sunae Seo for their comments and discussions.

References

[ANS88] ANSI/IEEE Std 1076–1987. *IEEE Standard VHDL Language Reference Manual.* New York, USA, 1988.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proc. of the International Conference on Formal Methods in Programming and their Applications, Lecture Notes in Computer Science 735*, pages 128–141. Springer-Verlag, 1993.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(2–3):103–179, 1992.

[CFR+99] Edmund M. Clarke, Masahiro Fujita, Sreeranga P. Rajan, Thomas W. Reps, Subash Shankar, and Tim Teitelbaum. Program slicing of hardware description languages. In *Conference on Correct Hardware Design and Verification Methods*, pages 298–312, 1999.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages*, pages 84–96, Tucson, Arizona, January 1978.

[Gra97] Philippe Granger. Static analysis of congruence properties on rational numbers (extended abstract). In *SAS 97, volume 1302. Lecture Notes in Computer Science*, 1997.

[HDT87] Susan Horwitz, Alan J. Demers, and Tim Teitelbaum. An efficient general iterative algorithm for dataflow analysis. *Acta Informatica*, 24(6):679–694, 1987.

[Hym01] C. Hymans. Modular analysis of a circuit description language by abstract interpretation and application to the automatic extraction of circuit shapes. Master's thesis, LIX, École Polytechnique, 2001.

[INIY96] M. Iwaihara, M. Nomura, S. Ichinose, and H. Yasuura. Program slicing on vhdl descriptions and its applications. In *Asian Pacific Conference on Hardware Description Languages (APCHDL)*, pages 132–139, 1996.

[Kar76] Michael Karr. Affine relationships among variables of a program. *Acta Informatica*, 6:133–151, May 1976.

[Min01] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[Plo81] G. D. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, University of Aarhus, 1981.

An Embedded Language Framework for Hardware Compilation

Koen Claessen[1] and Gordon Pace[2]

[1] Chalmers University, Gothenburg, Sweden
[2] INRIA Rhône-Alpes, Grenoble, France

Abstract. Various languages have been proposed to describe synchronous hardware at an abstract, yet syntesisable level. We propose a uniform framework within which such languages can be developed, and combined together for simulation, synthesis, and verification. We do this by embedding the languages in Lava — a hardware description language (HDL), itself embedded in the functional programming language Haskell. The approach allows us to easily experiment with new formal languages and language features, and also provides easy access to formal verification tools aiding program verification.

1 Introduction

There are two essentially different ways of describing hardware. One way is *structural* description, where the designer indicates what components should be used and how they should be connected. Designing hardware at the structural level can be rather tedious and time consuming. Sometimes, one affords to exchange speed or size of a circuit for the ability to design a circuit by describing its behaviour at a higher level of abstraction which can then be automatically *compiled* down to structural hardware. This way of describing circuit is usually called a *synthesisable behavioural description*[1]. Behavioural descriptions are also often used to describe the specification of a circuit.

There exist a number of languages that one can use to structurally describe hardware. An example is the synchronous language Lustre [8, 9], which can be compiled into hardware structurally [21]. Languages that can be used for synthesisable behavioural description are for example Esterel [2] and Occam [17]. The popular industrial description languages VHDL and Verilog allow both kinds of descriptions.

In this paper, we will only deal with synchronous hardware, that is, all latches in a circuit listen to one omnipresent global clock. Moreover, at every clock cycle, if each input to a circuit is defined, each point in the circuit stabilises to exactly

[1] These are to be distinguished from *behavioural descriptions* (as used in industrial HDLs such as Verilog and VHDL) which are used to describe the functionality of a circuit, but are do not necessarily have a hardware counterpart.

one voltage, low or high. However, we do not require that every feedback loop in the circuit contains a latch.

There are two main classes of synthesisable languages: ones where the description determines the timing behaviour (cycle by cycle) of the resultant circuit, and ones with no explicit timing control, and where the compilation only guarantees that the output at the end of the algorithm (or at designated points in the algorithm) matches that of the circuit. Languages with strict timing are necessary to describe circuits such as protocol implementations, and reactive systems, where the circuit continuously runs, sampling inputs, and behaving accordingly. In practice, some compilation schemata fall somewhere in between these two classes. In particular, commercial synthesis tools for Verilog and VHDL usually provide the user with the option of choosing how strictly the timing behaviour specified is adhered to. In the rest of the paper, we will be talking *exclusively* of strict timing compilation, but the approach is equally applicable to languages with loose timing.

Embedded Description Languages

Using a technique from the programming language community, called *embedded languages* [11], we present a framework to merge structural and behavioural hardware descriptions. An embedded description language is realised by means of a library in an already existing programming language, called the *host language*. This library provides the syntax and semantics of the embedded language by exporting function names and implementations.

The basic embedded language we use is *Lava* [5]. Lava is a structural hardware description language embedded in the functional programming language Haskell [18]. From hardware descriptions in Lava, EDIF netlist descriptions can be automatically generated, for example to implement the described circuit on a Field Programmable Gate Array (FPGA). This has previously led to highly efficient implementations of complicated circuits [6, 25].

Embedding a language is a powerful concept because descriptions in the embedded language are first-class objects in the host language. In the case of Lava, this means that hardware descriptions can be generated, analysed and transformed using a full-blown programming language.

The idea is now to build a layer on top of Lava, which embeds a synthesisable behavioural description language. In order to do this, we have to specify the syntax of the behavioural language, and the way it is compiled into a structural hardware description. It is possible to describe all this in the Lava framework: the syntax is described as a Haskell datatype, and the compilation process described as a Lava circuit description.

But why stop there? It is possible to embed several different behavioural description languages, each with their own features, advantages and disadvantages. In this way, we can describe a hardware system, using different languages for different parts, all within a single framework.

Examples of uses of embedding in this way are: behavioural in structural, where we use a behavioural language to describe some parts, and plug these parts together using a structural language; multiple behavioural in structural, the same, but having several different behavioural languages; structural in behavioural, so that we can describe a sub-procedure of the behavioural algorithm structurally; and even behavioural in behavioural, where we can describe sub-procedures for one behavioural language by using another behavioural language. All these examples are useful in describing circuits as well as their specifications.

Some of these examples are non-trivial to achieve, and we do not claim to have a generic solution to them. Our contribution proposes a common framework, in which one can quickly experiment with different approaches and new behavioural languages. The framework we propose, Lava, is powerful enough to use for describing new languages, giving semantics to them, implementing them, and combining them. In the context of developing behavioural description languages, it is very convenient to have circuit descriptions, analyses, transformations, and implementation and verification methods backed up by a full-blown programming language.

In section 2 we briefly introduce Lava and show how a simple high level language, that of regular expressions, can be embedded in Lava and how instances of this language can then be manipulated syntactically and compiled into circuits. In section 3 we illustrate how the embedded language approach extends easily to more complex languages by presenting a small, imperative style language, Flash. Section 4 then discusses more advanced issues: various ways of combining different high level languages, verification of compiled programs and exploring potentially dangerous combinational loops.

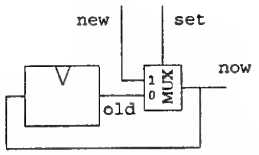
2 Embedding Hardware Description Languages

Circuit Descriptions in Lava

Circuit descriptions in Lava correspond to function definitions in Haskell. The Lava library provides primitive hardware components such as gates, multiplexers and delay components. We give a short introduction to Lava by example.

Here is an example of a description of a register. It contains a multiplexer, `mux`, and a delay component, `delay`. The delay component holds the state of the register and is initialised to `low`.

```
setRegister (set, new) = now
  where
    old = delay low now
    now = mux (set, (old, new))
```



Note that `setRegister` is declared as a circuit with two inputs and one output. Note also that definitions of outputs (`now`) and possible local wires (`old`) are given in the `where`-part of the declaration.

After we have made a circuit description, we can simulate the circuit in Lava as a normal Haskell function. We can also generate VHDL or EDIF describing the circuit. It is possible to apply circuit transformations such as retiming, and to perform circuit analyses such as performance and timing analysis. Lava is connected to a number of formal verification tools, so we can also automatically prove properties about the circuits.

Generic and Parametrized Circuit Definitions

We can use the one bit register to create an n-bit register array, by putting n registers together. In Lava, inputs which can be arbitrarily wide are represented by means of lists. A generic circuit, working for any number of inputs, can then be defined by recursion over the structure of this list.

```
setRegisterArray (set, [])       = []
setRegisterArray (set, new:news) = val:vals
  where
    val  = setRegister (set, new)
    vals = setRegisterArray (set, news)
```

Note how we use pattern matching to distinguish the cases when the list is empty (`[]`) and non-empty (`x:xs`, where `x` is the first element in the list, and `xs` the rest).

Circuit descriptions can also be parametrized. For example, to create a circuit with n delay components in series, we introduce n as a parameter to the description.

```
delayN 0 inp = inp
delayN n inp = out
  where
    inp' = delay low inp
    out  = delayN (n-1) inp'
```

Again, we use pattern matching and recursion to define the circuit. Note that the parameter `n` is *static*, meaning that it has to be known when we want to synthesise the circuit.

A parameter to a circuit does not have to be a number. For example, we can express circuit descriptions which take other circuits as parameters. We call these parametrized circuits *connection patterns*. Other examples of parameters include truth tables, decision trees and state machine descriptions. In this paper, we will talk about circuit descriptions which take behavioural hardware descriptions, or *programs*, as parameters.

Behavioural Descriptions as Objects

In order to parametrize the circuit definitions with behavioural descriptions, we have to embed a behavioural description language in Lava. We do this by declaring a Haskell datatype representing the syntax of the behavioural language. To illustrate the concepts with a small language, we will use regular expressions. The syntax of regular expressions is expressed as a Haskell datatype:

```
data RegExp = EmptyString
            | Input Sig
            | Star RegExp
            | RegExp :+: RegExp
            | RegExp :>: RegExp
```

The data objects belonging to this type are interpreted as regular expressions with, for example, $a(b+c)^*$ being expressed as:

```
Input a :>: Star (Input a :+: Input c)
```

Note that the variables `a`, `b` and `c` are of type `Sig` — they are *signals* provided by the programmer of the regular expression. They can either be outputs from another existing circuit, or be taken as extra parameters to the definition of a particular regular expression. We interpret the signal `a` being high as the character 'a' being present in the input.

Since regular expressions are now simply data objects, we can generate these expressions using Haskell programs. Thus, for example, we can define a `power` function for regular expressions:

```
power 0 e = EmptyString
power n e = e :>: power (n-1) e
```

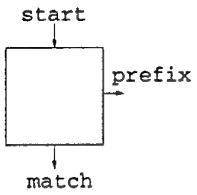
Similarly, regular expressions can be manipulated and modified. For example, a simple rewriting simplification can be defined as follows:

```
simplify (EmptyString :>: e) = simplify e
simplify (EmptyString :+: e)
  | containsEmpty e          = simplify e
  | otherwise                = EmptyString :+: simplify e
simplify (Star (Star e))     = simplify (Star e)
...
```

Another useful algorithm which can be expressed is the one presented in [20], which reduces (in linear time) a regular expression e to another one f such that the empty string does not occur in f and e^* is the same language as f^*. Thus, from now on, we assume that the body of a `Star` cannot produce the empty string.

Compiling Regular Expressions into Circuits

The circuits we generate for regular expressions have one input `start` and two outputs `match`, and `prefix`. When `start` is set to high, the circuit will start sampling the signals. The output `match` is then set to high when the resulting sequence of signals is included in the language represented by the expression. The output `prefix` corresponds to a wire which indicates whether the compiled circuit is still active, and the parsing of the regular expression has not yet failed with respect to the received inputs. Note that the circuit will get extra inputs, which correspond to the parsed symbols. They are part of the regular expression, by means of the `Input` construct.

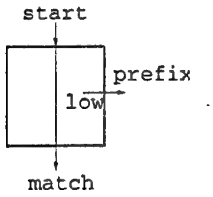


The type of the resulting circuit is thus:

```
type Circuit_RegExp = Sig -> (Sig, Sig)
```

since the resulting circuit has one input and two outputs. We express the compilation process as a circuit definition parametrized by a regular expression:

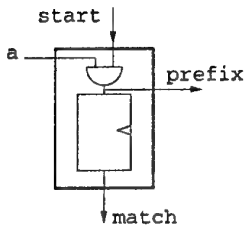
```
regexp :: RegExp -> Circuit_RegExp
```



The Empty String

The compilation of the empty string is straightforward, given the usage of the `prefix` and `match` wires:

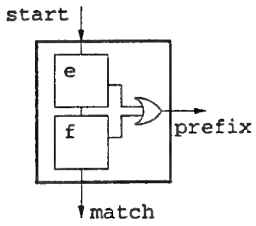
```
regexp EmptyString start = (prefix, match)
 where
    prefix = low
    match  = start
```



Signal input

The regular expression `Input a` is matched if, and only if the signal `a` is high when the circuit is started.

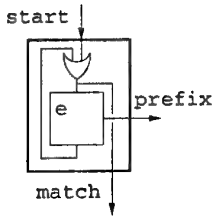
```
regexp (Input a) start = (prefix, match)
  where
    prefix = and2 (start, a)
    match  = delay low prefix
```



Sequential composition

The regular expression `e :>: f` must start accepting expression `e`, and upon matching it, start trying to match expression `f`.

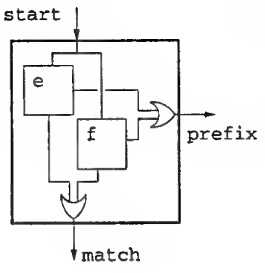
```
regexp (rexp1 :>: rexp2) start = (prefix, match)
  where
    (prefix1, match1) = regexp rexp1 start
    (prefix2, match)  = regexp rexp2 match1
    prefix = or2 (prefix1, prefix2)
```



Loops

The circuit accepting regular expression `Star e` is very similar to that accepting `e`, but it is restarted every time the inputs match `e`.

```
regexp (Star rexp) start = (prefix, match)
  where
    (prefix, match') = regexp rexp match
    match = or2 (start, match')
```



Non-deterministic choice
The inputs match regular expression `e :+: f` exactly when they match expression `e` or `f`.

```
regexp (rexp1 :+: rexp2) start = (prefix, match)
  where
    (prefix1, match1) = regexp rexp1 start
    (prefix2, match2) = regexp rexp2 start
    prefix = or2 (prefix1, prefix2)
    match  = or2 (match1, match2)
```

A circuit resulting from such a compilation scheme is not necessarily efficient enough. Often, there are optimisations we can make, such as constant folding (when the input to a gate is always low or always high), sharing introduction (when we have identical gates with identical inputs), tree introduction (changing a linear chain of associative gates into a balanced tree), and constant introduction (when a circuit point provably always has the same value). Sometimes, more rigorous optimisation methods are necessary; in this case we can use external circuit optimisation tools such as SIS [7].

3 Compiling Flash

In this section, we will show a slightly bigger example of a language, we will call *Flash*. It is quite a basic language, but it illustrates many of the issues one encounters when dealing with hardware compilation. As it is meant just an example, we deal quite informally with the semantics of Flash. More formal treatment of the semantics of similar languages can be found in [1, 17].

Flash Syntax
As before, we first declare a Haskell datatype that embeds the syntax of Flash.

```
data Flash = Skip
           | Delay
           | Shout
           | IfThenElse Sig (Flash, Flash)
           | While Sig Flash
           | Flash :>> Flash
           | Flash :|| Flash
```

Flash is a simple imperative programming language containing the usual statements like skip, sequential composition (`:>>`), if-then-else, and while. For simplicity, the language has no expressions. Instead, we can use Lava gates directly to create a signal representing the condition in both the if-then-else and the while loop.

To create some interesting output, we have added a `Shout` statement. This statement is in the spirit of the Esterel `emit` statement [2]. It makes a special output

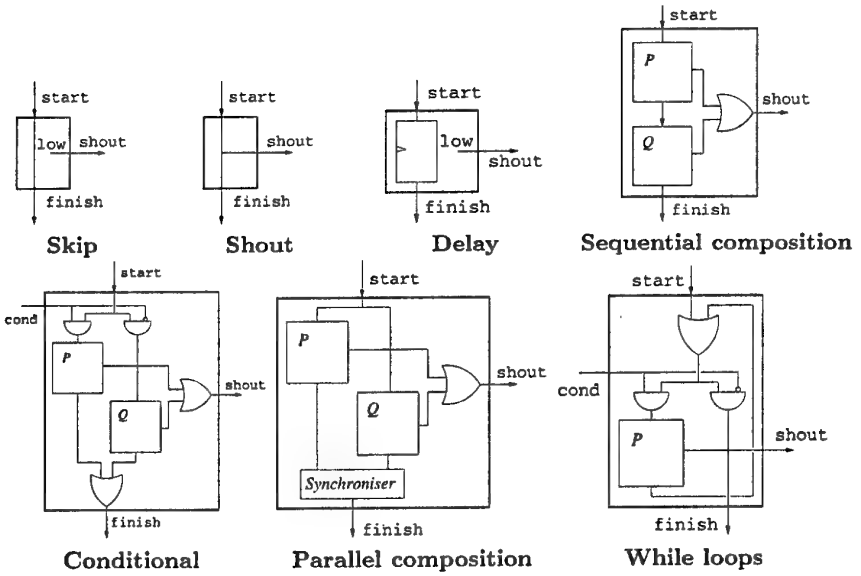


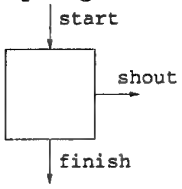
Fig. 1. Compiling Flash

of the circuit, called `shout`, high whenever `Shout` is executed. Further, we also have parallel composition (`:||`), which has a fork-join semantics. Lastly, the delay statement is the only statement that takes time. When executed, it blocks the process until the next clock cycle. Note that `Shout` takes no time to execute.

For example, a Flash program to output a clock-like output alternating between high and low could be written as:

```
alternate = While (high) (Shout :>> Delay :>> Delay)
```

Compiling Flash



The circuits that we compile Flash programs into have one input, `start`, which is set to high to start the program. They will have two outputs: `shout`, which becomes high when the program shouts, and `finish`, which becomes high when the program is done.

In figure 1, we see the compilation schemata for the various language constructs of Flash. We show the Lava code for some of the constructs.

The case for the while loop looks as follows:

```
flash (While cond prog) start = (shout, finish)
  where
    (shout, finish') = flash prog start'
```

```
restart = or2 (start, finish')
start'  = and2 (restart, cond)
finish  = and2 (restart, inv cond)
```

We might (re)start the body of the while loop, if the whole loop is started or if the body has just finished. In that case, depending on the condition, we restart the body or we finish. Note that we have created a loop since `finish'` depends on `start'` depends on `restart` depends on `finish'`. In fact, this loop might be a combinational loop — we say more about this in section 4.

Here is how we translate parallel composition:

```
flash (prog1 :|| prog2) start = (shout, finish)
  where
    (shout1, finish1) = flash prog1 start
    (shout2, finish2) = flash prog2 start
    shout  = or2 (shout1, shout2)
    finish = synchroniser (finish1, finish2)
```

We start both processes as soon as the parallel composition is started. We shout when one of the processes shouts. But when do we finish? We use a little circuit, called *synchroniser*, which keeps track of both processes, and generates a high on the finish signal exactly when both processes have finished.

```
synchroniser (finish1, finish2) = finish
  where
    both   = and2 (finish1, finish2)
    one    = xor2 (finish1, finish2)
    wait   = delay low (xor2 (one, wait))
    finish = or2 (both, and2 (wait, one))
```

The wire `both` is high when both processes are finishing at the same time. The wire `one` is high when exactly one process is finishing. The wire `wait` is high when one process has finished but not the other.

4 Advantages of Embedding

In this section, we discuss some of the advantages of embedding behavioral languages in a general hardware description framework like Lava.

Combining Languages
The choice of the right language to solve a problem is crucial both to simplify the algorithm, and to generate more efficient circuits. For example, regular expressions can be very useful to generate circuits which validate their input, but, since they have no outputting mechanism, it becomes very difficult (or impossible) to perform calculations and output their results.

Consider the problem of designing a circuit that accepts input sequences that behave like a clock with half-period n. This circuit might be useful for monitoring real input, or when expressing properties for later formal verification. It is easy to write a generic regular expression with the specified behaviour:

```
acceptClock n c = Star (  power n (Input c)
                       :>: power n (Input (inv c))
                       )
```

Now consider using a regular expression to design a circuit that monitors two inputs, accepting them only if they behave like clocks with half-periods n and m. The size of the smallest regular expression capable of doing this has a size of the order of magnitude of the least common denominator of n and m, which is too big in practice.

There are two solutions. One is to design a new language, in which it is easy to describe circuits as the one mentioned above. In fact, it would suffice to add *conjunction* as a regular expression operator, which would require some extra compile-time effort. The other is to combine the solutions to the two subproblems (recognising each clock) at the structural level using Lava:

```
acceptTwoClocks n m (c1,c2) = ok
  where
    (ok1,_) = regexp (acceptClock n c1) start
    (ok2,_) = regexp (acceptClock m c2) start
    start   = delay high low
    ok      = and2 (ok1, ok2)
```

Obviously, the used subprograms need not be in the same language. For example, if we want to run a Flash program `prg` only to abort it as soon as the input does not match a regular expression `rexp`, we can use the following parameterised circuit:

```
abort rexp prg start = (shout', finish)
  where
    (shout, finish) = flash  prg  start
    (prefix, _)     = regexp rexp start
    shout'          = and2 (shout, prefix)
```

Nesting Languages

A problem with the approach mentioned above is that we deal with the input and output of the produced circuits at a rather low-level. This is quite error-prone, and it becomes difficult to change the shape of the produced circuits.

A cleaner approach is not to express the combination of programs at the structural level, but at the behavioural one. Thus, for example, one could allow adding Flash subprograms to regular expressions by augmenting the syntax of regular expressions by:

```
data RegExp = ...   | ImportFlash Flash
```

Consider the problem of generating a circuit which recognises the input of `a`, `b` and `c` in any order. If this is required in a sub-expression of a regular expression, the result of expanding the expression can lead to a blow up in circuit size. However, a Flash program for this is rather simple to write:

```
wait s        = While (inv s) Delay
perm3 (a,b,c) = (wait a :|| wait b :|| wait c) :>> Delay
```

If this is required within a regular expression, one can easily use it as for example:

```
Star (ImportFlash (perm3 (a,b,c)) :+: ImportFlash (perm3 (d,e,f)))
```

Fiddling with the interfaces to make them match is thus done only once when the compilation of a regular expression of the form `ImportFlash p` is defined. However, this approach still has the undesirable effect that for every new language one uses, the compilers for all other languages need to be modified to be able to import programs from the new languages into the old ones.

A more extendable approach would be to add one `Import` construct for each language:

```
data RegExp = ...   | ImportRegExp Circuit_RegExp
data Flash  = ...   | ImportFlash  Circuit_Flash
```

Now, in order to import Flash programs in regular expressions, all we have to provide is a parameterised circuit `flash_regexp`, which converts from one format to the other.

```
flash_regexp flashc start = (prefix, match)
  where
    (shout, finish) = flashc start
    prefix = shout
    match  = finish
```

Needless to say, there are other ways in which a Flash circuit can be transformed into one which can be used by regular expressions. For example, one can generate (or calculate) an `active` wire from Flash circuits which corresponds to the regular expression `prefix` wire. In defining these 'conversion' circuits, we have to be careful here not to invalidate the invariants that the languages involved assume and obey. The technique mentioned in the next section can be used to help with this. 'Calling' another language now simply becomes a matter of using the `Import` construct and the right conversion circuits.

Error Wires

Often, something can go wrong during the execution of a program. What exactly can go wrong depends on the semantics of the language. A standard example in a language with arithmetic expressions is division by zero. It is not clear what

the corresponding compiled circuit would do in that case, since we do not want the circuit simply to 'abort'.

In a language with parallel composition, things can go wrong due to parts of the circuit requiring single access: two processes trying to send a message on the same channel at the same time, two processes updating a shared variable at the same time, etc. If the semantics of the language disallows these situations, then we should make sure that the programs we compile to hardware are well-behaved.

The solution we propose is to have an extra output to the circuit which goes high as soon as something goes wrong with the program execution — an *error* wire. This wire and the logic generating it will not appear in the final implementation of the circuit, but will be used to *verify* (by means of model checking methods) that the program in question is error-free.

Consider a change to the semantics of Flash, requiring that only one process can shout at the same time. We would like to be warned at compile time if a program violates that property. Thus, we add an error wire to the output of Flash circuits, and adapt the compilation scheme accordingly. Here is the interesting case, parallel composition:

```
flash (prog1 :|| prog2) start = (shout, error, finish)
  where
    (shout1, error1, finish1) = flash prog1 start
    (shout2, error2, finish2) = flash prog2 start
    shout  = or2 (shout1, shout2)
    both   = and2 (shout1, shout2)
    error  = orl [error1, both, error2]
    finish = synchroniser (finish1, finish2)
```

There is an error in parallel composition of two programs if there was an error in (at least) one of the processes, or if both processes shout at the same time. We can now declare a *property*, a circuit which outputs are always high if and only if a certain property holds.

```
prop_FlashProgramOk prog start = inv error
  where
    (_, error, _) = flash prog start
```

The output ok is high if and only if there is no error in the program prog. We can check this property using the Lava command verify.

```
Lava> verify (prop_FlashProgramOk (alternate :|| (Delay :>> alternate))
Verify: ... Valid.

Lava> verify (prop_FlashProgramOk (Shout :>> Delay :|| Shout))
Verify: ... Falsifiable.
<high>
```

The error wire technique can also be used to find bugs in the compilation scheme itself. Many languages have certain invariants that hold for every program. By raising the signal on the error wire when the invariant is violated, and verifying the absence of this error for random programs (by using a technique similar to the one developed in [4]), we can find bugs or increase our confidence in the compilation scheme.

Combinational Loops

Looking at the compilation scheme for the `while` construct, we can see that it is possible to introduce combinational loops: cycles in the circuit without a delay component.

The usual solution in this case is to require that body of the `while` loop takes time — the execution path goes through at least one `Delay` statement. But even with this restriction, the resulting circuit might still contain combinational loops. However, these combinational loops are not *bad*, in the sense that the actual circuit never produces undefined outputs. In this case, the combinational loops are called *constructive* [24].

Even when all combinational loops in a given circuit are constructive, most of the external formal verification tools that Lava is connected to, are not able to deal with these loops. Fortunately, the method of *temporal induction* [23] can naturally verify properties of cyclic circuit definitions. However, the method is only sound if all loops in the circuit are constructive loops.

Thus, before we implement or formally verify actual circuits containing possible bad loops, we have to prove that all loops all constructive. Lava provides a circuit analysis, called `constructive`, which does exactly that [3]. Here is how we can use it:

```
Lava> verify (constructive (flash (While high Delay)))
Verify: ... Valid.

Lava> verify (constructive (flash (While high Skip)))
Verify: ... Falsifiable.
<high>
```

What about parallel composition? When is it acceptable for a body of a while loop to contain a parallel composition? Take, for example, the following Flash program:

```
possibleProblem inp = While high
                        ( IfThenElse inp (Skip , Delay)
                       :|| Delay
                        )
```

In principle, we should be able to execute this, since for all programs `p`, the program `p :|| Delay` takes time to execute. Let us analyse the resulting circuit:

```
possibleProblemCirc inp = flash (possibleProblem inp)

Lava> verify (constructive possibleProblemCirc)
Verify: ... Falsifiable.
<low, high>
<high, low>
```

This shows that the simple compilation scheme we have used to illustrate our examples is not sufficiently robust to handle this example. Obviously, one can require that both sides of a parallel composition should take time (when appearing immediately inside the body of a while loop). However, this is a stringent and rather unsatisfactory restriction. A better solution would be to use a more complex compilation of loops, as used, for example, in [1].

5 Conclusions

Related Work

Hardware compilation of high-level languages has been around for quite a while. The approach has been considered potentially practical mainly since the introduction of programmable circuits. The compilation for various languages have since appeared in the literature, see e.g. [15, 17, 16, 1]. An introductory overview of the methodology appears in [26].

It is widely recognised that different styles of synchronous languages lend themselves more easily to different applications. In [13, 14], Maraninchi and Rémond present Mode-Automata — a combination of state diagram based descriptions (based on Argos [12]) with the dataflow language Lustre [8]. The semantics of the resulting language are defined by a translation into plain Lustre. The approach is thus very similar to the one we use, except that they use external programs to read mode-automata and translate them into Lustre. The embedded language approach we use, allows us to translate and reason about the new language at the same level as our base HDL Lava. This allows a much more versatile approach to language combination.

Poigné and Holenderski [19] present a theoretical framework for combining synchronous languages by using synchronous automata as the common semantic level. These ideas have been implemented in the SYNCHRONY WORKBENCH where programs written in one of a number of languages (Esterel, Lustre, Argos, and Synchronous Eifel) can be combined together. The main difference between their work and that presented in this paper, is that we embed the languages we use, and our intermediate language, Lava, is itself an embedded language. This gives us certain advantages: it is easier to add new languages to the framework, and language combination can be easily adapted depending on the requirements.

Discussion

We have presented a uniform framework in which it is easy to implement and

hence experiment with synthesisable behavioural languages. By embedding these languages in Lava, we are able to define their compilation in a natural and easy way and, at the same time, benefit from the verification tools connected to Lava to improve compilation and verify programs. We also benefit from, the fact that we can directly generate EDIF netlists or VHDL from the circuits generated by our embedded compilers.

Using this approach, we have shown that we can formally reason about programs at a number of levels. First, taking advantage of the fact that our programs are just data objects in Haskell, we can apply syntactic reasoning by defining functions which modify the program. Second, using the verification tools linked to Lava, we can define observers (in one of the languages) to verify properties of hardware described using either structural Lava, or some other language. Third, the compilation process itself can make use of the verification tools to check dynamic properties which may be needed to guarantee correct compilation.

Within our framework, we have implemented various languages or subsets of them, such as Esterel, Handel and Occam, fragments of process calculi such as CSP and CBS, and some restricted temporal logics. We have also embedded state machine descriptions and specification languages in the same framework. This included different control and data features including updatable variables, buffered and unbuffered channels, exceptions and broadcast communication. Describing the compilation of a language is rather straightforward, and in fact, we have successfully used this framework in the teaching of a graduate course on hardware description languages. The definition of the compilation function for a language is usually not much different from a denotational semantics of the language in terms of a dataflow network.

One of the important issues that we have not discussed in this paper is the question of the correctness of the compilation procedure. A number of approaches have been proposed [10, 22, 1] which are applicable to our compilation scheme. We are currently exploring how such proofs can also be presented uniformly within our framework. Preliminary work is encouraging and it is not difficult to prove that, for instance, the compilation of regular expressions presented in this paper satisfies regular expression equational axioms.

References

1. Gérard Berry. The constructive semantics of Pure Esterel. Unfinished draft, available from `http://www.esterel.org`, 1999.
2. Gérard Berry. The Esterel primer. Available from `http://www.esterel.org`, 2000.
3. K. Claessen. Safety property verification of cyclic circuits. In preparation, 2002.
4. K. Claessen and J. Hughes. QuickCheck: A light-weight tool for random testing of Haskell programs. In *Internatioanl Conference on Functional Programming*, 2000.
5. K. Claessen and M. Sheeran. A tutorial on Lava: A hardware description and verification system. Available from `http://www.cs.chalmers.se/~koen/Lava`, 2000.
6. Koen Claessen, Mary Sheeran, and Satnam Singh. The design and verification of a sorter core. In *CHARME*. Springer, 2001.

7. E. M. Sentovich and K. J. Singh et al. SIS: A system for sequential circuit synthesis. Technical Report Berkeley, UCB/ERL M92/41, 1992.
8. N. Halbwachs. A tutorial of Lustre. Available from `http://www-verimag.imag.fr/SYNCHRONE`, 1993.
9. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language Lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.
10. Jifeng He, Ian Page, and Jonathan Bowen. Towards a provably correct hardware implementation of Occam. In G.J. Milne and L. Pierre, editors, *Correct Hardware Design and Verification Methods*, number 683 in LNCS. Springer, 1993.
11. Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys*, 28(4):196, 1996.
12. F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR*, number 630 in LNCS. Springer, 1992.
13. F. Maraninchi and Y. Rémond. Compositionality criteria for defining mixed-styles synchronous languages. In *International Symposium: Compositionality – The Significant Difference*, number 1536 in LNCS. Springer, 1997.
14. Florence Maraninchi and Yann Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*. Springer, 1998.
15. David May. Compiling Occam into silicon. In C. A. R. Hoare, editor, *Developments in Concurrency and Communication*, University of Texas at Austin Year of Programming Series, chapter 3, pages 87–106. Addison-Wesley, 1990.
16. Ian Page. Constructing hardware-software systems from a single description. *Journal of VLSI Signal Processing*, 12(1):87–107, 1996.
17. Ian Page and Wayne Luk. In Wayne Luk and Will Moore, editors, *FPGAs*.
18. Simon Peyton Jones and John Hughes et al. Report on the programming language Haskell 98, a non-strict, purely functional language. Available from `http://haskell.org`, 1999.
19. A. Poigné and L. Holenderski. On the combination of synchronous languages. In *International Symposium: Compositionality – The Significant Difference*, number 1536 in LNCS, pages 490–514. Springer, 1997.
20. Pascal Raymond. Recognizing regular expressions by means of dataflow networks. In *23rd International Colloquium on Automata, Languages, and Programming, (ICALP'96)*, number 1099 in LNCS. Springer, 1996.
21. F. Rocheteau and N. Halbwachs. Pollux, a Lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, 1991.
22. M. Schenke and M. Dossis. Provably correct hardware compilation using timing diagrams. Available from `http://semantik.Informatik.Uni-Oldenburg.DE/persons/michael.schenke/`, 1997.
23. Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a SAT-solver. In *FMCAD*, LNCS 1954. Springer, 2000.
24. T. Shiple, G. Berry, and H. Touati. Constructive analysis of cyclic circuits. In *European Design and Test Conference*, 1996.
25. Satnam Singh and Phil James-Roxby. Lava and JBits: From HDL to bitstream in seconds. In K.L. Pocek and J.M. Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*. IEEE Computer Society Press, 2001.
26. Niklaus Wirth. Hardware compilation: Translating programs into circuits. *Computer*, 31(6):25–31, 1998.

Modelling with Streams in Daisy and The SchemEngine Project

Steven D Johnson
Indiana University Computer Science Department
System Design Methods Laboratory

sjohnson@cs.indiana.edu
www.cs.indiana.edu/~sjohnson

Reviewers' comments

"I would like to see a retrospective on Daisy, summarizing the lessons learned in designing the language and especially interfacing to ordinary hardware design flows, all illustrated by a real-ish example like SchemEngine, and topped off by some ideas for new application areas (e.g. embedded software). I think the combination of 14/15 would serve this purpose well, if the author is willing to take things in this direction."

"Not clear what ideas the use of streams in Daisy has compared to other languages (like Lava) modelling data as streams. What's the role of laziness and lazy cons apart from streams?"

"... I would like the author to concentrate on paper 15. (Why have these been merged? They address separate unrelated issues, if I have understood this correctly."

Outline

I. *Background and context*

II. *Modeling with streams in Daisy*

A. Lazy Cons demand-oriented computation, stream I/O, concurrency.

B. Stream systems.

C. Some modeling techniques.

D. Distributed extensions.

III. *The* SchemEngine *Project*

A. Design derivation

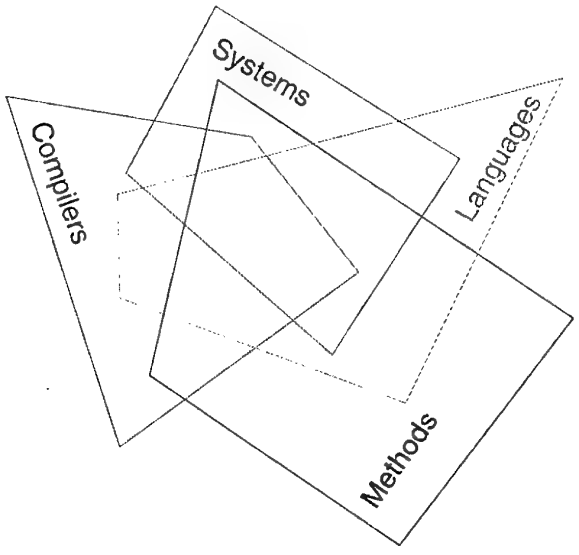
B. Previous studies, *Schemachine*

C. *VLISP* [Guttman, Ramsdell, Wand, *L&SC 95]*

D. *SchemEngine* objectives in integrated formal analysis

E. Toward an integrated codesign enviornment

I. Background and context, 1975–now



Background (languages)

75-80 Functional programming languages
Fridman & Wise, "CONS should not evaluate its arguments"
Applicative programming for systems
Extensions for indeterminacy, `set`: $[\alpha\ \beta\ \cdots]$
Suspending construction model [continuations, engines in Scheme]
Semantics ?!

80-85 Daisy/DSI in Unix
Stream-based I/O From concurrency to parallelism
O'Donnell, programming environments, hardware models

85-90 Parallel DSI [Jeschke95]
Language-driven architecture ⇒ design derivation

90-95 Bounded speculation
Windows on the data space
Daisy/DSI for small-scale MIMD?

95-00 Distributed demand propagation
HW models as case studies

Background (methods)

75-80 Functional programming methods
Prosser & Winkel, structured digital design, ASMs

80-85 Compiler derivation [Wand]
Combinator factorization ⇒ "machine"
Stream systems and synchronous hardware
Formalized synthesis

85-90 Digital design derivation
Functional/algebraic formalism
GC-PLD, GC-VLSI, SECD

90-95 DDD ⇒ ⋯ [Bose, Tuna, Rath, W.Hunt]
Heterogeneous reasoning
FM8502, FM9001-DDD, Schemachine
DDD vis-a-vis PVS, coinductive types [Minor]

95-00 Tools
Behavior tables
etc.

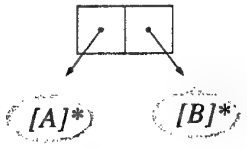
II. Modeling with streams in Daisy

Animation is a key aspect of functional formalism.

- Suspending CONS, demand oriented computation.
- List (stream) representation of I/O
- Concurrency construct, set
- Windowing support

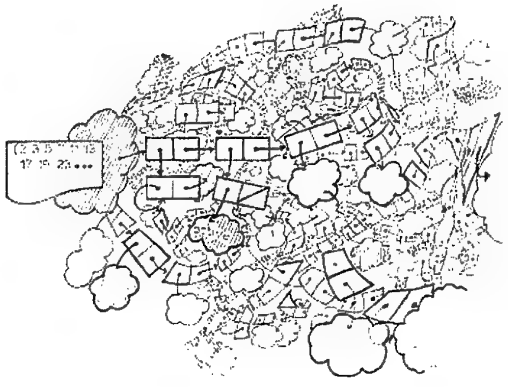
Suspending CONS

Delays? No.
Futures? No.
Engines? Almost.
Demand driven computation? No.
Demand oriented computation ···
bounded speculation

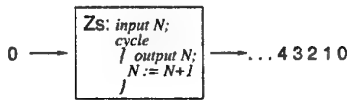


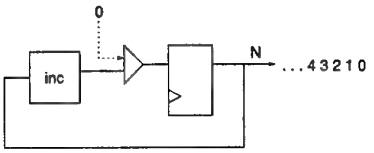
DSI:

- ◇ Heap based symbolic multiprocessing
- ◇ Transparent process management



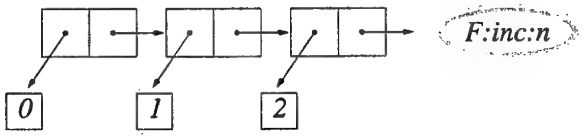
Processes as streams





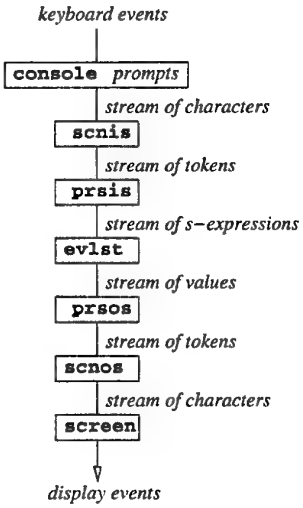

```
F:n = [n !  F:inc:n]
```

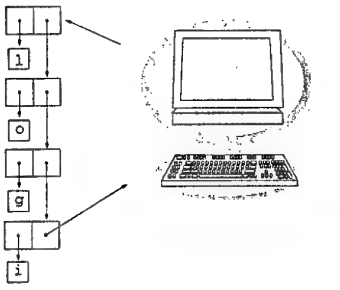
```
N = [0 !  (map:inc):N]
```



Stream (i.e. lazy-list) based I/O

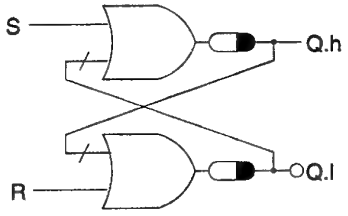
I/O synchronization and suspension coercion use the same synchronization mechanism (e.g. a *presence bit*)



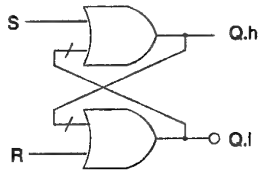


`screen:console:`*prompt*

```
NOT = (map:not)
OR = (map:xps:or)


RSFF = \[S R].
 rec
   Qh = [0 ! OR:[S NOT:Ql]]
   Ql = [1 ! OR:[S NOT:Qh]]
  in
   [Qh Ql]


bit-filter = \[C ! Cs].
   if:[ same?:[C "0"] [0 ! bit-filter:Cs]
        same?:[C "1"] [1 ! bit-filter:Cs]
                      bit-filter:Cs
      ]


xps:
RSFF:[bit-filter:console:"NL S: "
      bit-filter:console:"NL R: "]
```




```
& xps:
RSFF:[bit-filter:console:"NL S: "
      bit-filter:console:"NL R: "]


[
S: 0 0 0 0 0 0 0 0
[1 0] [
R: 0 0 0 1 1 1 0 0
 0] [1 0] [1 0] [1 1] [0 1] [0 1] [0 1]
S: 0 0 0 1 1 1 0 0 0
 [0 1] [0
R: 0 0 0 0 0 0 0 0 0
 1] [0 1] [0 1] [1 1] [1 0] [1 0] [1 0] [1 0]
S:
```

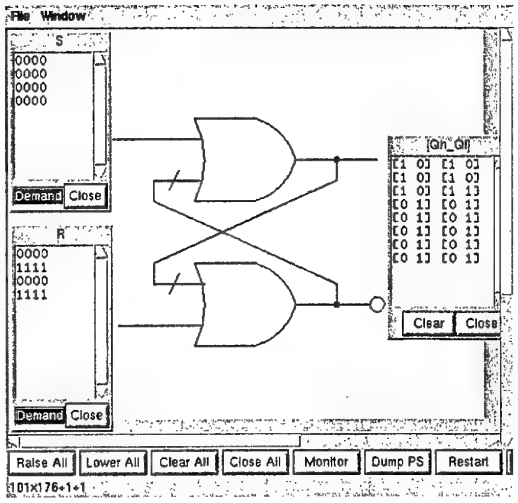


Widget devices:

`wndi`: *name* → *char**

`wndo`: [*name*, *char**] → []

Also: `filei/o`,
`socketi/o*`,
`execi/o*`, ...




```
wndo:[ "[Qh Ql]"
       prsos:
        scnos:
         xps:
          RSFF:[ bitfilter:wndi:"S"
                 bitfilter:wndi:"R" ]]
```

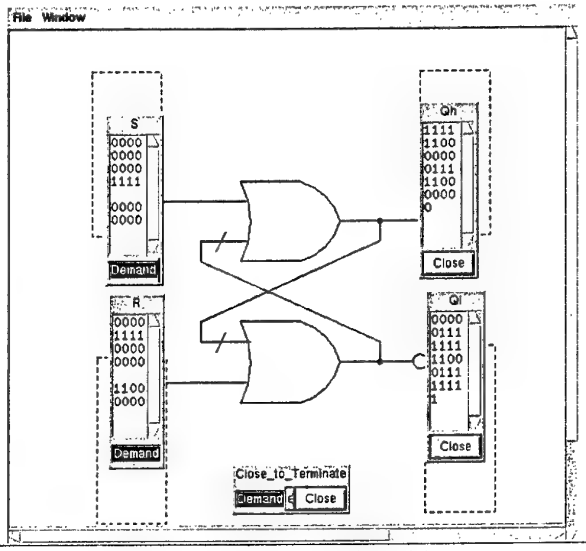

Concurrency

Implicit through bounded speculation

Explicit through (constructs such as) `set`

- Notation: expression A, computation α, result a.
- `set`: $[A\ B\ C]$ constructs list object $L = [\alpha\ \beta\ \gamma]$
- L becomes manifest as $[a\ b\ c]$, or $[b\ a\ c]$, or $[b\ c\ a]$, *etc.*,
- There is an imprecise operational relationship with computational effort.
- *Semantics (?!)*

```
let [Qh Ql] = RSFF:[ bit-filter:wndi:"S"
                     bit-filter:wndi:"R" ]
in  set:[ wndo:["Qh" bit_to_chr:Qh]
          wndo:["Ql" bit_to_chr:Ql]
          consume:wndi:"Click_CLOSE_to_terminate"   ]
```



More techniques, examples

- Time stamping prompts
- Synchronizing windows
- Dataflow
- Asynchronous interactions, `merge` and `split`
- Distributed modeling; sockets and pipes
- Scripting?

III. The SchemEngine Project

Objectives [Johnson, IUTR 544]

- Advancing design derivation
- System-level formal analysis
- From semantics to hardware
- Heterogeneous reasoning
- Embedded applications (!?)
- Foundations for high-confidence

DDD studies in language-driven architecture

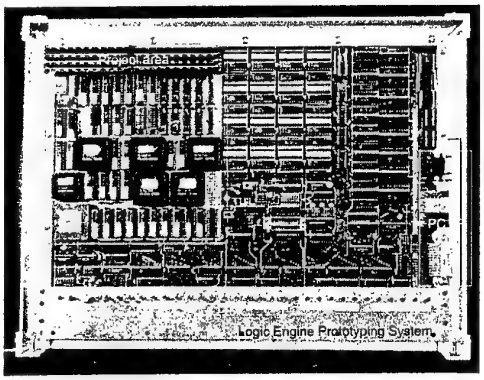
`www.cs.indiana.edu/hmg/`

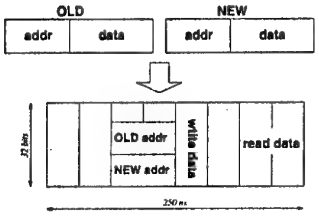
- Garbage collectors in PLDs, VLSI, FPGAs [Boyer 86-90]
- SECD computer [Wehrmeister 89]
- Schemachine [Burger 94]

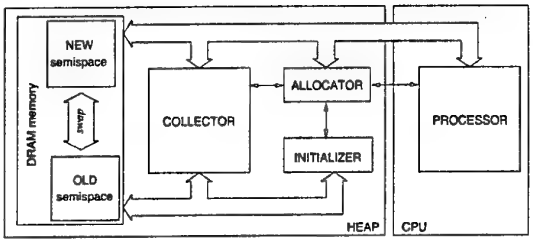
The bigger picture

- Compiler correctness [Wand, Clinger 80-85]
- Compiler *derivation* [Wand 80-85]
- Scheme based methodology and pedagogy
- Codesign tools

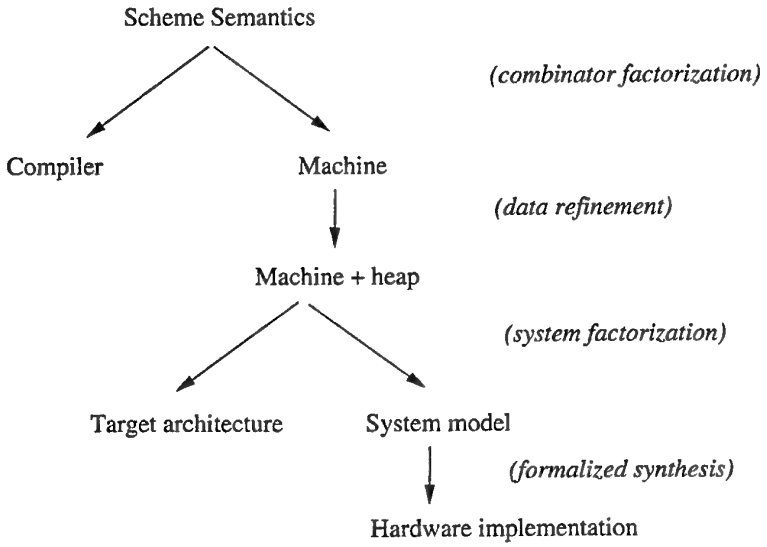
- CPU, GC, ALLOC, INIT derived with DDD
- CPU is naive
- Memory system tuned to GC
- PLDs, mux-based FPGAs, DRAM simms.
- Codesign using Scheme and Logic Engine
  - Spec. models
  - Derived models
  - Staging to hardware







Relationshop to VLISP [Guttman, Ramsdell, Wand, L&SC 95]



Pending issues

- Retargetting Schemachine (Virtex, SDRAM?), cores
- Advancing CPU design
- Verification of system-level properties
- Behavior table case study
- Synchronization abstractions
  - Hierarchical clocks
  - Bisimulation modulo protocols
  - Interface abstraction
  - Software factorization
- Algorithmic correctness proofs
- Integrating modeling, derivation, and co-design
- Close with VLISP
- Embedded resource management
- Java

Functional Design using Behavioural and Structural Components

Richard Sharp
rws26@cl.cam.ac.uk

University of Cambridge Computer Laboratory
William Gates Building
JJ Thomson Avenue
Cambridge CB3 0FD, UK

Abstract

In previous work we have demonstrated how the functional language SAFL can be used as a *behavioural* hardware description language. Other work (such as μFP and Lava) has demonstrated that functional languages are apposite for *structural* hardware description.

One of the strengths of systems such as VHDL and Verilog is their ability to mix structural- and behavioural-level primitives in a single specification. Motivated by this observation, we describe a unified framework in which a stratified functional language is used to specify hardware across different levels of abstraction: Lava-style structural expansion is used to generate acyclic combinatorial circuits; these combinatorial fragments are composed at the SAFL level. We demonstrate the utility of this programming paradigm by means of a realistic case-study. Our tools have been used to specify, simulate and synthesise a DES encryption/decryption circuit. Area-time performance figures are presented.

1 Introduction

Hardware description languages (HDLs) are often categorised according to the level of abstraction they provide. *Behavioural HDLs* focus on algorithmic specification and attempt to abstract as many low-level implementation issues as possible. Most behavioural HDLs support constructs commonly found in high-level programming languages (e.g. assignment, sequencing, conditionals and iteration). In contrast, *Structural HDLs* allow a hardware engineer to describe a circuit by specifying its hardware-level components and their interconnections. The process of automatically translating a Behavioural HDL into a Structural HDL is often referred to as *high-level synthesis*.

Commercially the two most important HDLs are Verilog and VHDL [8, 7]. A contributing factor to the success of these systems is their support for *both* behavioural *and* structural-level design. The ability to combine behavioural and structural primitives in a single specification offers engineers a powerful framework: when the precise low-level details of a component are not critical, behavioural constructs can be used; for components where finer-grained control is required, structural constructs can be used.[1] However, the flip-side is that by supporting multiple levels of abstraction both Verilog and VHDL are very large languages which are difficult to analyse, transform and reason about.

In previous work we have designed SAFL [11], a *behavioural* HDL which supports a functional programming style. An optimising high-level synthesis system has been implemented which compiles SAFL specifications into structural Verilog [8]. (We map the generated Verilog to silicon using commercially available RTL compilers.) Other researchers have demonstrated that functional languages are powerful tools for *structural* hardware specification [19, 14, 3]. In this paper

[1] Note the analogy with embedding assembly code in a higher-level software language.

we present a system which integrates both structural- and behavioural-level hardware design in a pure functional framework. Our technique involves embedding a functional language designed for structural hardware description into SAFL.

The remainder of this paper is structured as follows. After surveying related work (Section 2) we give a brief overview of the SAFL language (Section 3). Our mechanism for embedding Lava-style structural expansion in SAFL is then presented (Section 4). This methodology is demonstrated by means of a realistic case-study in which a fully functional DES encrypter/decrypter is specified (Section 5).

2 Related Work

There is a large body of work on using functional languages to describe hardware at the structural level. Notable systems in this area include μFP [19], HDRE/Hydra [14], Hawk [9] and Lava [3]. The central idea behind each of these systems is to use the powerful features found in existing functional languages (e.g. higher-order functions, polymorphism and lazy evaluation) to build up netlists from simple primitives. These primitives can be given different semantic interpretations allowing, for example, the same specification to be either simulated or translated into a netlist. However, whilst this technique is obviously appealing, there are problems involved in generating netlists for circuits which contain feedback loops. The difficulty is that, in a pure functional language, a cyclic circuit (expressed as a series of mutually recursive equations) naturally evaluates to an infinite tree preventing the netlist translation phase from terminating.

A number of solutions to this problem have been proposed: O'Donnell advocates the explicit tagging of components at the source-level [15]. In this system the programmer is responsible for labelling distinct components of a circuit with unique values. Whilst this allows a pure functional graph traversal algorithm to detect cycles trivially (by maintaining a list of tags which have already been seen) it imposes an extra burden on the programmer and significantly increases potential for manual error (since it is the programmer's job to ensure that distinct components have unique tags). Lava [3] also uses tagging to identify cycles, but employs a *state monad* [20] to generate fresh tags automatically. Although this neatly abstracts the low-level tagging details from the designer, Claessen and Sands argue that the resulting style of programming is "unnatural" and "inconvenient" [5]. In the same paper, Claessen and Sands propose another solution which involves augmenting Haskell (the functional language in which Lava is embedded) with immutable references which support a test for equality. This extension makes graph sharing observable at the source-level but, although it is shown that many useful laws still hold, full equational reasoning is no longer possible—for example, β-reduction no longer preserves equality.

In this paper we present an alternative approach. By only allowing the description of *acyclic* circuits through Lava-style structural static expansion and then combining these circuit fragments at the SAFL level we facilitate the *pure functional* specification of complex circuits which can contain feedback loops. We have not solved the observable sharing problem; instead we have eliminated it: since cycles are not permitted at the structural level we do not have to worry about infinite loops being statically expanded. Conversely, since feedback loops are represented as tail-recursive calls at the SAFL-level there is no need to introduce impure language features.

Although most of the work on using functional languages for hardware description focuses on the *structural* level some researchers have considered using functional languages for *behavioural* hardware description. Johnson's Digital Design Derivation (DDD) system [4] uses a scheme-like language to describe circuit behaviour. A series of semantics-preserving transformations are presented which can be used to refine a behavioural specification into a circuit structure; the transformations are applied manually by an engineer. This is a different approach to hardware design using SAFL [11] where, although semantics-preserving transformations are used to explore architectural tradeoffs (including allocation, binding and scheduling [6]) at the source-level, the resulting SAFL specification is fed into an optimising compiler which generates a structural hardware design automatically.

3 Overview of the SAFL Language

SAFL has syntactic categories e (term) and p (program). First suppose that v ranges over a set of constants. Let x range over variables (occurring in `let` declarations or as formal parameters), a over primitive functions (such as addition) and f over user-defined functions. For typographical convenience we abbreviate formal parameter lists $(x_1, \ldots, x_k)$ and actual parameter lists $(e_1, \ldots, e_k)$ to $\vec{x}$ and $\vec{e}$ respectively; the same abbreviations are used in `let` definitions. Then the abstract syntax of the core SAFL language can be given in terms of recursion equations over programs, p, and expressions, e:

$$\begin{array}{rcl} e & ::= & v \mid x \mid \texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 \mid \texttt{let } \vec{x} = \vec{e} \texttt{ in } e_0 \mid \\ & & a(e_1, \ldots, e_{arity(a)}) \mid f(e_1, \ldots, e_{arity(f)}) \\ p & ::= & \texttt{fun } f_1(\vec{x}) = e_1 \ \ldots \ \texttt{fun } f_n(\vec{x}) = e_n \end{array}$$

It is sometimes convenient to extend this syntax slightly. In later examples we use a `case`-expression instead of iterated tests; we also write `e[n:m]` to select a bit-field `[n..m]` from the result of expression `e` (where `n` and `m` are integer constants).

There is a syntactic restriction that whenever a call to function f_j from function f_i is part of a cycle in the call graph of p then we require the call to be a tail call.[2] (Note that calls to a function not forming part of a cycle can occur in an arbitrary expression context.) This ensures that storage for the variables and temporaries of p can be allocated statically—in software terms the storage is associated with the code of the compiled function; in hardware terms it is associated with the logic to evaluate the function body.

The other main feature of SAFL, apart from static allocatability, is that its evaluation is limited only by data flow (and control flow at user-defined call and conditional). Thus, in the form $\texttt{let } \vec{x} = (e_1, \ldots, e_k) \texttt{ in } e_0$ or in a call $f(e_1, \ldots, e_k)$ or $a(e_1, \ldots, e_k)$, all the e_i $(1 \leq i \leq k)$ are evaluated concurrently. In the conditional $\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3$ we first evaluate (only) e_1; one of e_2 or e_3 is evaluated after its result is known. SAFL has call-by-value semantics since eager evaluation offers a greater opportunity for parallelism (i.e. we can execute a function call's arguments in parallel without worrying about strictness).

Although up to this point we have referred to SAFL as a behavioural language, it is also capable of capturing some structural aspects of a design. We say that SAFL is *resource-aware* to indicate that a single user-defined function definition at the source-level corresponds to a single hardware resource at the circuit-level. In this context multiple calls to the same function corresponds to resource sharing[3]. We use SAFL-level transformations to express architectural tradeoffs such as resource duplication/sharing and hardware/software co-design [12]. In essence these transformations preserve a specification's *extensional* semantics (the result returned) whilst changing the *intensional* semantics (how the circuit is structured). A more in-depth description of the SAFL language and its associated silicon compiler can be found in our recent survey paper [13]. For the purposes of this document we provide a short example which illustrates the main points:

```
fun mult(x:16, y:16, acc:32):32 =
  if (x=0 | y=0) then acc
     else mult(x<<1, y>>1, if y[0:0] then acc+x else acc)

fun f(x:16):32 = mult(x, x, 0) + mult(13, x, 0)
```

From this specification, two hardware resources are generated: a circuit, $H_{\texttt{mult}}$, corresponding to `mult` and a circuit, $H_{\texttt{f}}$, corresponding to `f`. The two calls to `mult` are not inlined: at the hardware level there is only one shared resource, $H_{\texttt{mult}}$, which is invoked twice by $H_{\texttt{cube}}$. The tail-recursive call in the definition of `mult` is synthesised into a feedback loop at the circuit level. Since function

[2] Tail calls consist of calls forming the whole of a function body, or nested solely within the bodies of `let-in` expressions or that are the consequents of `if-then-else` expressions.

[3] Our optimising compiler automatically deals with sharing issues by statically scheduling access to resources where it can, and generating arbiters to perform scheduling dynamically otherwise [18].

arguments are evaluated concurrently, the two shift operations occurring in the recursive call to `mult` are evaluated in parallel along with the conditional test and possibly, depending on the conditional branch taken, the addition operation.

Each SAFL variable is annotated with a bit-width at its point of introduction. We use the form `x:`w to indicate that variable `x` has width w. Note that the widths of function result types are also specified explicitly (using the form `fun f(...):`w). Widths of constants can either be specified explicitly or, more usually, inferred from their local context. As part of a simple type-checking phase our SAFL compiler ensures that for each function call, $f(\vec{x})$, the widths of arguments, $\vec{x}$ match those specified in the signature of f.

4 Embedding Structural Expansion in SAFL

Resource awareness allows SAFL to describe the *system-level* structure of a design by mapping `fun` declarations to circuit-level functional units. In contrast, systems such as μFP and Lava offer much finer-grained control over circuit structure, taking logic-gates (rather than function definitions) as their structural primitives. We are not arguing that either approach is better: in practice both are appropriate depending on the type of hardware that is being designed. Motivated by this observation, we present a framework which integrates Lava-style structural expansion with SAFL.

Section 4.1 outlines our system for fine-grained structural hardware description which, for the purposes of this paper, we will refer to as Magma[4]. In Section 4.2 we show how Magma is integrated with SAFL.

4.1 Building Combinatorial Hardware in Magma

An argument in favour of Lava, Hydra and other similar systems, is that since they are embedded in existing functional languages they are able to leverage existing tools and compilers. Furthermore, use of non-standard interpretation of basis functions means that the *same* compiler can be used to perform both hardware simulation and synthesis. These compelling benefits lead us to adopt a similar approach. However, in contrast to Lava, which is embedded in Haskell [1], we choose to embed Magma in ML [10]. The choice of ML is fitting for two main reasons: firstly, since we only wish to describe acyclic circuits, ML's strict evaluation is appropriate for both simulation and synthesis interpretations; secondly, since SAFL also borrows much of its syntax and semantics from ML, both Magma and SAFL share similar conventions (an important consideration when we are dealing with specifications containing a mixture of both Magma and SAFL).

4.1.1 An ML module system primer

In order to understand the workings of Magma some familiarity with the ML module system is required. Whilst we do not describe the full details of the ML-module system here, this section is sufficient to allow readers unfamiliar the module system to understand the remainder of this paper. For more information the reader is referred to a more in-depth survey [16].

The basic element of ML's module system is the `structure`. The structure provides a way of packaging both type and value (including function) definitions into a single entity. An important feature of structures is that they provide a hierarchical name-space. For example, if a function, f, is defined in a structure S we refer to it as $S.f$.

An ML `signature` provides a mechanism to specify interfaces. A signature contains a set of name and type-declarations. One can use a signature to constrain a structure using the ":" operator. Only values whose types are explicitly declared in the constraining signature are visible outside the constrained structure.

Finally, the ML module system provides `functors`. A functor is essentially a parameterised structure, dependent on another structure which is provided externally. For example, consider the

[4] As it is a restricted form of Lava.

```
signature BASIS =
    sig
        type bit
        val b0   : bit
        val b1   : bit
        val orb  : bit * bit -> bit
        val andb : bit * bit -> bit
        val notb : bit * bit
        val xorb : bit * bit -> bit
    end
```

Figure 1: The definition of the BASIS signature (from the Magma library)

following (contrived) code fragment which defines a functor, FTR, parameterised over a structure, S (where S is constrained by signature, SSIG):

```
functor FTR(S:SSIG) =
  struct
     val a = S.f(3)
  end
```

Passing a structure, T, into FTR yields a new structure containing a single item, a, which has the value T.f(3). Magma makes use of functors to parameterise hardware specifications over interpretations of their basis functions. This provides a convenient way of using the same code for both simulation and synthesis (see below).

4.1.2 Specifying Hardware in Magma

The Magma system essentially consists of a library of ML code. A signature called BASIS is provided which declares the types of supported basis functions (see Figure 1). Values b0 and b1 correspond to logic-0 (false) and logic-1 (true) respectively. Functions orb, andb, notb and xorb correspond to logic functions *or*, *and*, *not* and *xor*. Two structures which implement BASIS are provided:

- SimulateBasis provides a simulation interpretation. We implement bits as boolean values; functions orb, andb etc. have their usual boolean interpretations.
- SynthesisBasis provides a synthesis interpretation. We implement bits as strings representing names of wires in a net-list. Functions orb, andb etc. take input wires as arguments and return a (fresh) output wire. Calling one of the basis functions results in its netlist declaration being written to the selected output stream as a side-effect. For example, if the result of calling andb with string arguments "in_wire1" and "in_wire2" is the string "out_wire" then the following is output to StdOut:

```
and(out_wire,in_wire1,in_wire2);
```

Figure 2 shows a Magma specification of a ripple-adder. As with all Magma programs, the main body of code is contained within an ML functor. This provides a convenient abstraction, allowing us to parameterise a design over its basis functions. By passing in the structure SimulateBasis (see above) we are able to instantiate a copy of the design for simulation purposes; similarly, by passing in SynthesisBasis we instantiate a version of the design which, when executed, outputs its netlist. The signature RP_ADD is used to specify the type of the ripple_add function. Using this signature to constrain the RippleAdder functor also means that *only* the ripple_add function is externally visible; the functions carry_chain and adder can only be accessed from within the

```
signature RP_ADD =
  sig
      type bit
      val ripple_add : (bit list * bit list) -> bit list
  end

functor RippleAdder (B:BASIS):RP_ADD =
      struct

         type bit=B.bit
         fun adder (x,y,c_in) = (B.xorb(c_in, B.xorb(x,y)),
                                 B.orb( B.orb( B.andb (x,y), B.andb(x,c_in)),
                                        B.andb(y,c_in)))

         fun carry_chain f _ ([],[]) = []
           | carry_chain f c_in (x::xs,y::ys) =
                let val (res_bit, c_out) = f (x,y,c_in)
                 in res_bit::(carry_chain f c_out (xs,ys))
                end

         val ripple_add = carry_chain adder B.b0
      end
```

Figure 2: A simple ripple-adder described in Magma

functor. Note that the use of signatures to specify interfaces in this way is not compulsory but, for the usual software-engineering reasons, it is recommended.

Let us imagine that a designer has just written the ripple-adder specification shown in Figure 2 and now wants to test it. This can be done by instantiating a simulation version of the design in an interactive ML session:

```
- structure SimulateAdder = RippleAdder (SimulationBasis);
```

The adder can now be tested by passing in arguments (a tuple of `bit` lists) and examining the result. For example:

```
- SimulateAdder.ripple_add ([b1,b0,b0,b1,b1,b1],[b0,b1,b1,b0,b1,b1])
val it = [b1,b1,b1,b1,b0,b1]  : SimulateAdder.bit list
```

Let us now imagine that the net-list corresponding to the rippler-adder is required. We start by instantiating a synthesis version of the design:

```
- structure SynthesiseAdder = RippleAdder (SynthesisBasis);
```

If we pass in lists of input wires as arguments, the `ripple_add` function prints its netlist to the screen and returns a list of output wires:

```
- SynthesiseAdder.ripple_add (Magma.new_bus 5, Magma.new_bus 5)
and(w_1,w_45,w_46);
and(w_2,w_1,w_44);
...
and(w_149,w_55,w_103);
val it = ["w_149","w_150","w_151","w_152","w_153"]
```

The function `new_bus`, part of the Magma library, is used to generate a bus of given width (represented as a list of wires).

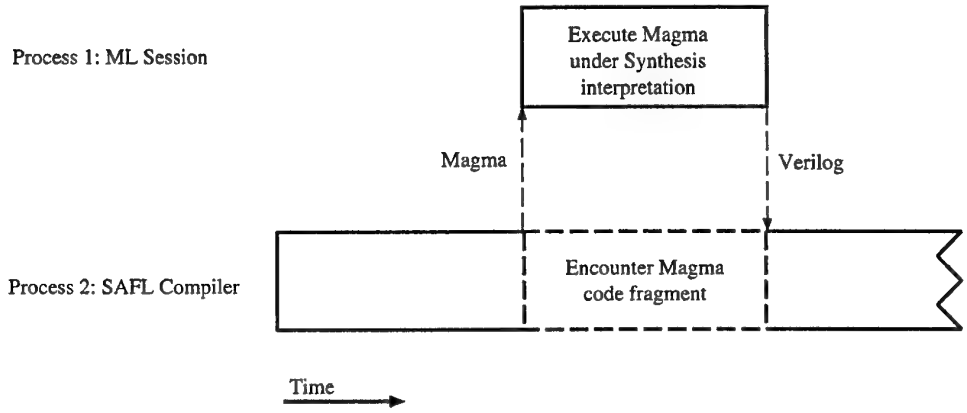


Figure 3: A diagrammatic view of the steps involved in compiling a SAFL/Magma specification

```
(* Magma library block containing Magma_Code functor: *)

<%
  signature RP_ADD =
     ... (* as in Figure 2 *)

  functor Magma_Code (B:BASIS):RP_ADD =
     ... (* as RippleAdder functor in Figure 2 *)
%>

fun mult(x, y, acc) =
  if (x=0 | y=0) then acc
    else mult(x<<1, y>>1,
              if y[0] then <% ripple_add %>(acc,x) else acc)
```

Figure 4: A simple example of integrating Magma and SAFL into a single specification

4.2 Integrating SAFL and Magma

Our approach to integrating Magma and SAFL involves using delimiters to embed Magma code fragments inside SAFL programs. At compile time the embedded Magma is synthesised and the resulting netlist is incorporated into the generated circuit (see Figure 3). This technique was partly inspired by web-scripting frameworks such as ASP and PHP [2] which can be embedded in HTML documents[5]. To highlight this analogy we use ASP-style delimiters "<%" and "%>" to mark the start and end points of Magma code fragments. Our compiler performs simple width checking across the SAFL-Magma boundary, ensuring the validity of the final design.

The SAFL parser is extended to allow a special type of Magma code fragment at the beginning of a specification. This initial Magma fragment, which is referred to as the *library block*, contains an ML functor called `Magma_Code`. Functions within `Magma_Code` can be called from other Magma fragments in the remainder of the specification. Figure 4 illustrates these points with a simple example in which the Magma ripple adder (initially defined in Figure 2) is invoked from a SAFL specification. The precise details of the SAFL-Magma integration are discussed later in this section; for now it suffices to observe that Magma fragments are treated as functions at the SAFL-level and applied to SAFL expressions.

[5]When a dynamic web-page is fetched the ASP or PHP code is executed generating HTML which is returned to the client.

The treatment of Magma fragments is similar to that of primitive functions (such as +, -, * etc.). In particular, Magma code fragments are expanded in-place. For example, if a specification contains two Magma fragments of the form, <% ripple_add %>, then the generated hardware contains two separate ripple adders. Note that if we require a shared ripple_adder then we can encapsulate the Magma fragment in a SAFL function definition and rely on SAFL's resource-awareness properties. For example, the specification:

```
fun add(x, y) = <% ripple_add %> (x,y)
fun mult_3(x) = add(x, add(x,x))
```

contains a single ripple adder shared between the two invocations within the definition of the mult_3(x) function. Since embedded Magma code fragments represent pure functions (i.e. do not cause side effects) they do not inhibit SAFL-level program transformation. Thus our existing SAFL-level transformations corresponding to resource duplication/sharing [11], hardware/software co-design [12] etc. remain valid.

Implementation and Technical Details

Consider the general case of a Magma fragment, m, embedded in SAFL:

$$\texttt{<\%}\ m\ \texttt{\%>}(e_1, \ldots, e_k)$$

where $e_1, \ldots, e_k$ are SAFL expressions. On encountering the embedded Magma code fragment, <% m %>, our compiler performs the following operations:

1. An ML program, $\mathcal{M}$, (represented as a string) is constructed by concatenating the *library block* together with commands to instantiate the Magma_Code functor in its synthesis interpretation (see above).

2. The bit-widths of SAFL expressions, $e_1, \ldots, e_k$, are determined (bit-widths of variables are known to the SAFL compiler) and ML code is added to $\mathcal{M}$ to construct corresponding busses, $B_1, \ldots, B_k$, of the appropriate widths (using the Magma.new_bus library call).

3. $\mathcal{M}$ is further augmented with code to:

   (a) execute ML expression, $m(B_1, \ldots, B_k)$, which, since the library block has been instantiated in its synthesis interpretation, results in the generation of a netlist; and

   (b) wrap up the resulting netlist in a Verilog module declaration (adding Verilog wire declarations as appropriate).

4. A new ML session is spawned as a separate process and program $\mathcal{M}$ is executed within it.

5. The output of $\mathcal{M}$, a Verilog module declaration representing the compiled Magma code fragment, is returned to the SAFL compiler where it is added to the object code. Our SAFL compiler also generates code to instantiate the module, connecting it to the wires corresponding to the output ports of SAFL expressions $e_1, \ldots, e_k$.

In order that the ML-expression $m(B_1, \ldots, B_k)$ type checks, m must evaluate to a function, $\mathcal{F}$, with a type of the form:

```
(bit list * bit list * ... * bit list)  ->  bit list
```

with the arity of $\mathcal{F}$'s argument tuple equal to k. If m does not have the right type then a type-error is generated in the ML-session spawned to execute $\mathcal{M}$. Our SAFL compiler traps this ML type-error and generates a meaningful error of its own, indicating the offending line-number of the SAFL/Magma specification. In this way we ensure that the bit-widths and number of arguments applied to <% m %> at the SAFL-level match those expected at the Magma-level.

Another property we wish to ensure at compile time is that the output port of a Magma-generated circuit is of the right width. We achieve this by incorporating width information corresponding to the output port of Magma-generated hardware into our SAFL compiler's type-checking phase. Determining the width of a Magma specification's output port is trivial—it is simply the length of the `bit` list returned when $m(B_1, \ldots, B_k)$ is executed.

5 Case Study: DES Encrypter/Decrypter

Appendix A presents code fragments from the SAFL specification of a Data Encryption Standard (DES) encryption/decryption circuit. Here we describe the code for the DES example, focusing on the interaction between SAFL and Magma; the details of the DES algorithms are not discussed. We refer readers who are interested in knowing more about DES to Scheier's cryptography textbook [17].

The library block at the beginning of the DES specification defines three functions used later in the specification:

- `perm` is a curried function which takes a permutation pattern, p, (represented as a list of integers) and a list of bits, l. It returns l permuted according to pattern p.
- `ror` is a curried function which takes an integer, x, and a list of bits, l. It returns l rotated right by x.
- `rol` is as `ror` but rotates bits left (as opposed to right).

A set of permutation patterns required by the DES algorithm are also declared. (For space reasons the bodies of some of these declarations are omitted.)

The code in Appendix A uses two of SAFL's features which have not been described in this paper:

- The primitive function `join` takes an arbitrary number of arguments and returns the bit-level concatenation of these arguments. As one would expect, the bit-width of the result of a call to `join` is the sum of the bit-widths of its input arguments.
- SAFL's `type` declaration allows us to construct records with named fields. Curly braces, { ... }, are used as record constructors and dot notation ($r.f$) is used to select a field, f, from record r. After type-checking our SAFL compiler translates record notation directly into bit-level `join`s and selects. (Recall that bit-level selects are represented using the `e[n:m]` notation—see Section 3.)

Primitive functions corresponding to arithmetic and boolean operators use their standard symbols (e.g. +, <, =). The binary infix operator, (^), is used for bit-wise exclusive-or.

The DES algorithm requires 8 S-boxes, each of which is a substitution function which takes a 6-bit input and returns a 4-bit output. The S-boxes' definitions make use of one of SAFL's syntactic sugarings:

`lookup` e `with` $\{v_0, \ldots, v_k\}$

Semantically the `lookup` construct is equivalent to a `case` expression:

`case` e `of 0 =>` v_0 `|` $\ldots$ `|` $(k-1)$ `=>` v_{k-1} `default` v_k.

To ensure that each input value to the `lookup` expression has a corresponding output value we enforce the constraint that $k = 2^w - 1$ where w is the width of expression e. Our compiler is often able to map `lookup` statements directly into ROM blocks, leading to a significantly more efficient implementation than a series of iterated tests.

Before applying its substition each S-box permutes its input. We use our Magma permutation function to represent this permutation: `<% perm p_inSbox %>(x)`. Other examples of SAFL-Magma integration can be seen throughout the specification. The `keyshift` function makes use

of the Magma `ror` and `rol` functions to generate a key schedule. Other invocations of the Magma `perm` function can be seen in the bodies of SAFL-level functions: `round` and `main`. We find the use of higher-order Magma functions (such as `perm`, `ror` and `rol`) to be a powerful idiom.

We used our tools to map the DES specification to synthesisable RTL-Verilog. A commercial RTL-synthesis tool (Leonardo from Exemplar) was used to map synthesise the RTL-Verilog for a Xilinx Virtex-V300 FPGA. The resulting circuit utilised less than 5% of the FPGA's resources and could be clocked at 70MHz. Since encrypting/decrypting a block of data takes 17 cycles, a throughput of 260 Megabits/sec can be achieved.

6 Conclusions and Further Work

In this paper we have motivated and described a technique for combining both behavioural and structural-level hardware specification in a stratified pure functional language. Our methodology has been applied to a realistic example. We believe that the major advantages of our approch are as follows:

- As in Verilog and VHDL, we are able to describe large systems consisting of both behavioural and structural components.
- SAFL-level program transformation remains a powerful technique for architectural exploration. The functional nature of the Magma-integration means that our existing library of SAFL transformations are still applicable.
- By only dealing with combinatorial circuits at the structural-level we eliminate the problems associated with graph-sharing in a pure functional language (see Section 2). We do not sacrifice expressivity: cyclic (sequential) circuits can still be formed by composing combinatorial fragments at the SAFL-level in a more controlled way.

In future work we would like to investigate using similar techniques for integrating systems such as Lava or Magma directly into Verilog. Whilst this would not give the formal benefits of our pure functional SAFL-Magma hybrid, it may help to make the tried-and-tested technique of embedding a structural hardware specification using functional languages more accessible to engineers working in industry.

Acknowledgements

This work was supported by (UK) EPSRC grant, reference GR/N64256: "A Resource-Aware Functional Language for Hardware Synthesis"; AT&T Research Laboratories Cambridge provided additional support (including sponsoring the author). The author would like to thank Alan Mycroft for his valuable comments and suggestions.

Appendix A: SAFL specification of a DES encrypter/decrypter

```
(* Magma Library block *)

<%
signature DES =
  sig
     val perm: int list -> 'a list -> 'a list
     val ror:  int -> 'a list -> 'a list
     val rol:  int -> 'a list -> 'a list

     val p_compress : int list
     val p_key      : int list
      ...
     val p_inSbox   : int list
  end

functor Magma_code (B:BASIS):DES =
    struct

      (* DES permutation patterns ... *)

        val p_initial   = [58,50,42,34,26,18,10,2,60,52,44,36,28,20,12,4,
                           62,54,46,38,30,22,14,6,64,56,48,40,32,24,16,8,
                           57,49,41,33,25,17,9,1,59,51,43,35,27,19,11,3,
                           61,53,45,37,29,21,13,5,63,55,47,39,31,23,15,7]

        val p_key       = [57,49,41,33,25,17,9,1,58,50,42,34,26,18,
                           10,2,59,51,43,35,27,19,11,3,60,52,44,36,
                           63,55,47,39,31,23,15,7,62,54,46,38,30,22,
                           14,6,61,53,45,37,29,21,13,5,28,20,12,4]

        val p_pbox      = [16,7,20,21,29,12,28,17,1,15,23,26,5,18,31,10,
                           2,8,24,14,32,27,3,9,19,13,30,6,22,11,4,25]
        val p_compress  = [ ... <snip> ... ]
        val p_expansion = [ ... <snip> ... ]
        val p_final     = [ ... <snip> ... ]
        val p_inSbox    = [1,5,2,3,4]

     (* Higher-order permutation function -- given a list of bits
          and a pattern it returns a permuted list of bits: *)

        fun perm positions input =
            let val inlength = length input
                fun do_perm [] _ = []
                  | do_perm (p::ps) input =
                     (List.nth (input,inlength-p))::(do_perm ps input)
            in do_perm positions input
            end

      (* Rotate bits right by specified amount: *)
        fun ror n l =
            let val last_n = rev (List.take (rev l, n))
```

```
            val rest   = List.take (l, (length l)-n)
         in last_n @ rest
        end

    (* Rotate bits left by specified amount: *)

      fun rol n l =
        let val first_n = List.take (l, n)
            val rest    = List.drop (l, n)
        in rest @ first_n
        end

  end
%>

(* Definitions of S-Boxes (implemented as simple lookup tables) *)

fun sbox1(x:6):4 =
  lookup <% perm p_inSbox %> (x)
         with {14,4,13,1,2,15,11,8,3,10,6,12,5,9,0,7,
               0,15,7,4,14,2,13,1,10,6,12,11,9,5,3,8,
               4,1,14,8,13,6,2,11,15,12,9,7,3,10,5,0,
               15,12,8,2,4,9,1,7,5,11,3,14,10,0,6,13}

fun sbox2(x:6):4 =
  lookup <% perm p_inSbox %> (x)
         with { ... <snip> ... }

fun sbox3(x:6):4 = ...
    ...
fun sbox8(x:6):4 = ...

(* Do s_box substitution on data-block: *)

fun s_sub(x:48):32 =
  join( sbox1( x[47:42] ), sbox2( x[41:36] ),
        sbox3( x[35:30] ), sbox4( x[29:24] ),
        sbox5( x[23:18] ), sbox6( x[17:12] ),
        sbox7( x[11:6]  ), sbox8( x[5:0]   ))

(* Define a record which contains the left and right halves
   of a 64-bit DES block and the 56-bit key. *)

type round_data = record {left:32, right:32, key:56}

(*  Successive keys are calculated by circular shifts. The degree
    of the shift depends on the round (rd).
    We shift either left/right depending on whether we are
    decrypting/encrypting. Note that the inline pragma ensures that
    keyshift is never treated as a shared resource. *)

inline fun keyshift(key_half:28,rd:4,encrypt:1):28 =
  define val shift_one = (rd=0 or rd=1 or rd=8 or rd=15)
  in
```

```
    if encrypt then
        if shift_one then <% rol 1 %> (key_half)
                     else <% rol 2 %> (key_half)
    else
        if rd=0 then key_half
                else if shift_one then <% ror 1 %> (key_half)
                                  else <% ror 2 %> (key_half)
  end

(*  A single DES round:  *)

inline fun round(bl:round_data,rd:4,encrypt:1):round_data =
    let
         val lkey = keyshift(slice(bl.key,55,28),rd,encrypt)
         val rkey = keyshift(slice(bl.key,27,0),rd,encrypt)
         val keybits   = <% perm p_compress %> ( join(lkey,rkey) )
         val new_right = let val after_p = <% perm p_expansion %>(bl.right)
                         in s_sub (after_p ^ keybits ^ bl.left)
                         end

     in {left=bl.right, right=new_right, key=join(lkey,rkey)}
     end

(*  Do 16 DES rounds: *)

fun des(c:4, rd:round_data,encrypt:1):round_data =
  let
         val new_data = round(rd, c, encrypt)
  in if c=15 then new_data
             else des(c+1, new_data,encrypt)
  end

(*  Apply Key-Permutation to incoming 64 key bits,
    apply the initial permutation to incoming data-block,
    do 16-rounds of DES on permuted data-block,
    apply the final-permutation and return the encrypted block *)

fun main(block:64,key:64, encrypt:1):64 =
  let
         val block_p = <% perm p_initial %> (block)
         val realkey = <% perm p_key %> (key)
         val output = des(0:4, {left=slice(block_p,63,32),
                                right=slice(block_p,31,0),
                                key=realkey}, encrypt)

     in <% perm final %> (join(output.right, output.left))
     end
```

References

[1] Haskell98 report. Available from `http://www.haskell.org/`.

[2] PHP hypertext preprocessor. `http://www.php.net/`.

[3] BJESSE, P., CLAESSEN, K., SHEERAN, M., AND SINGH, S. Lava: Hardware description in Haskell. In *Proceedings of the 3rd International Conference on Functional Programming* (1998), SIGPLAN, ACM.

[4] BOSE, B. DDD: A transformation system for digital design derivation. Tech. Rep. 331, Indiana University, 1991.

[5] CLAESSEN, K., AND SANDS, D. Observable sharing for functional circuit description. In *Asian Computing Science Conference* (1999), pp. 62–73.

[6] DE MICHELI, G. *Synthesis and Optimization of Digital Circuits.* McGraw-Hill Inc., 1994.

[7] IEEE. Standard VHDL Reference Manual, 1993. IEEE Standard 1076-1993.

[8] IEEE. Verilog HDL language reference manual. IEEE Draft Standard 1364, October 1995.

[9] MATTHEWS, J., COOK, B., AND LAUNCHBURY, J. Microprocessor specification in Hawk. In *Proceedings of the IEEE International Conference on Computer Languages* (1998).

[10] MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML (Revised).* MIT Press, 1997.

[11] MYCROFT, A., AND SHARP, R. A statically allocated parallel functional language. In *Proceedings of the International Conference on Automata, Languages and Programming* (2000), vol. 1853 of *LNCS*, Springer-Verlag.

[12] MYCROFT, A., AND SHARP, R. Hardware/software co-design using functional languages. In *Proceedings of TACAS* (2001), vol. 2031 of *LNCS*, Springer-Verlag.

[13] MYCROFT, A., AND SHARP, R. Higher-level techniques for hardware description and synthesis. *To appear. International Journal on Software Tools for Technology Transfer (STTT)* (2002).

[14] O'DONNELL, J. Hardware description with recursion equations. In *Proceedings of the IFIP 8th International Symposium on Computer Hardware Description Languages and their Applications* (April 1987), North-Holland, pp. 363–382.

[15] O'DONNELL, J. Generating netlists from executable circuit specifications in a pure functional language. In *Functional Programming, Workshops in Computing, Proceedings* (1992), Springer-Verlag, pp. 178–194.

[16] PAULSON, L. *ML for the working programmer.* Cambridge University Press, 1996.

[17] SCHNEIER, B. *Applied cryptography: protocols, algorithms, and sourcecode in C.* John Wiley and Sons, New York, 1994.

[18] SHARP, R., AND MYCROFT, A. Soft scheduling for hardware. In *Proceedings of the 8th International Static Analysis Symposium* (2001), vol. 2126 of *LNCS*, Springer-Verlag.

[19] SHEERAN, M. muFP, a language for VLSI design. In *Proceedings of the ACM Symposium on LISP and Functional Programming* (1984).

[20] WADLER, P. Monads for functional programming. In *Advanced Functional Programming* (1995), vol. 925 of *LNCS*, Springer-Verlag.

A Proof Engine Approach to Solving Combinational Design Automation Problems

Gunnar Andersson, Per Bjesse, Byron Cook
Prover Technology
{guan,bjesse,byron}@prover.com

As a consequence of the fact that many combinational design automation problems are NP- or coNP-complete, one fundamental approach to their solution is to model the problems as propositional logic formulas, and apply some proof method to decide whether the formulas always evaluate to true.

However, there exists a plethora of individual techniques for discharging propositional proof conditions, including (but not limited to) the Davis-Putnam-Loveland-Logeman method, Stålmarck's method, Binary Decision Diagrams, and rewriting. Each proof technique has a particular characteristic in terms of space and time behaviour, sensitivity to the size of the system, and the particular domain it will work the best for. As a result, it is not unusual to combine techniques in different ways when solving particular classes of combinational design automation problems.

Unfortunately, there does not exist an individual composition of techniques that will be sufficient to deal with all problem instances at all future points in time. As the design climate changes, and the structure and complexity of the designs at hand change, every approach that is made up from a composition of individual methods will thus have to be modified accordingly.

At Prover Technology, we develop and maintain a plug-in proof engine that is used to solve combinational logic problems in design automation tools. This proof engine, PROVER CL, is delivered with default composite analyses for a number of domains.

As developers and maintainers of PROVER CL, we need to be able to experiment with new variations of the default analyses so that we can keep them updated. We also need to be able to develop new analyses that work well in new problem domains, and on problems with particular characteristics. It is thus very important that our proof engine framework is constructed in such a way that (1) it allows us to rapidly construct composite analyses with a minimum amount of effort, and (2) we can get the maximum amount of synergy between the individual methods.

In this talk, we present an approach to addressing these issues that Prover Technology has been working on for quite some time—initial results were already reported in early 2000 [CS00].

Our approach builds on two key ideas. First of all, we structure our proof engine in such a way that we view individual proof techniques as *strategies*; that is, functions between different proof states. Second, we define our strategies to

be compositional in the sense that they can be combined in any order to form more powerful proof-search strategies.

In this presentation, we introduce the ideas underlying our framework, and present how three popular proof techniques for propositional logic (Stålmarck's saturation method, the Davis-Putnam-Loveland-Logeman method, and BDDs) can be cast as strategies. We also present experimental results demonstrating that the synergy effects between our primitive strategies result in order of magnitude speedups compared to individual approaches such as the ZCHAFF search method, and state-of-the-art BDD-based analyses. The benchmark results come from two public benchmark suites, and one suite of inhouse problems from one of our industrial partners.

References

[CS00] Koen Claessen and Gunnar Stålmarck. A framework for propositional proof strategies. Technical report, Chalmers University of Technology, Computing Sciences Dept., January 2000. (Proceedings of the Departmental Winter Meeting).

State Abstraction Techniques for the Verification of Reactive Circuits

Yannis Bres[1], Gérard Berry[2], Amar Bouali[2], and Ellen M. Sentovich[3]

[1] CMA-EMP/INRIA (Yannis.Bres@cma.inria.fr)
[2] Esterel Technologies ([Gerard.Berry,Amar.Bouali]@esterel-technologies.com)
[3] Cadence Berkeley Labs (EllenS@cadence.com)

Abstract. Several techniques for formal verification of synchronous circuits depend on the computation of the reachable state space (RSS) of the circuit. Computing the exact RSS may be prohibitively expensive. In order to simplify the computation, the exact RSS can be replaced by an over-approximation of it, called the ORSS. The resulting verification computation will be conservative, and the larger the ORSS, the more conservative the approximation. A common technique for computing the ORSS is to replace some of its state variables by inputs. In this paper, we present a new approach based on variable abstraction using a three-valued logic. We also present a way to reduce the over-approximation by using structural information given by compilers of high-level languages like Esterel, ECL or SyncCharts. A real example of an avionic system is used to show the improvements that variable abstraction can bring.

1 Introduction

This paper deals with formal verification of synchronous designs derived from programs written either in Esterel [4, 5], ECL [15] or SyncCharts [1] languages. These languages are well suited for control-dominated programs, both for hardware and software targets. ECL and SyncCharts programs can be translated into Esterel. The Esterel compiler translates such programs into the pair of a sequential circuit and a data path.

Formal verification is currently performed on the control part of the program, by XEVE [2], a BDD-based verifier publicly available, or the verifier built-in the Esterel Studio tool [14]. The properties are expressed by *synchronous bug observers* [20], i.e. auxiliary signals that are emitted by the circuit in case of a safety property violation. Verification amounts to checking that observer signals can never be emitted. To check observers, XEVE uses a forward reachability technique well-adapted to Esterel control-dominated programs: it iteratively computes the reachable state space of the circuit, or *RSS*, checking at each step that observers cannot be emitted for any reachable state and any legal input. Although this approach has proved successful in handling quite large designs, it is limited by the potential explosion of BDDs during the computation of the RSS.

This paper is devoted to improvements of the verification algorithm based on *variable abstraction*. The global idea is to use over-approximations (ORSS) of the exact RSS, which is usually an overkill to prove safety properties. Verification using the ORSS is *conservative*: if a property is true for an ORSS, it is true for the original circuit, but a given ORSS may not prove the desired property. An ORSS can be obtained directly from the structure of the source program, as explained in [25], but its impact on verification performance is relatively limited. A better approach to simplify verification is to *reduce the number of variables and functions* occuring during the BDD computations. We study two techniques for this purpose: register inputization, in which a state variable is simply made free in the RSS, and register abstraction, in which we use a three-valued logic. Register inputization views some registers as free combinational variables, losing their state-holding contents. Register abstraction uses a three-valued logic and makes some register variables completely disappear from the BDD, which is attractive to improve computation times, but is a stronger abstraction. Both techniques can be combined with the aforementioned structural ORSS ones. We show the efficiency of our method on a real avionics system, the fuel management of a twin-engine jet aircraft from Dassault Aviation, and present two other experiments.

Section 2 presents the algorithms for RSS computation. Section 3 presents the ORSS abstraction techniques. Section 4 presents the application example, and section 5 concludes.

1.1 Related Work

One of the most studied approach to ORSS computation is based on FSM decomposition: in [11], Cho et al. proposed approximate RSS computation algorithms that decompose the set of state variables into disjoint subsets. Each subset is used to compute a portion of the RSS, and the cross-product is taken afterwards for an ORSS. Extension to non-disjoint subsets was described by Govindaraju et al. in [16], and refined in [17] through addition of auxiliary state variables that increase correlation between subsets. Such techniques perform *a posteriori* quantification, as state variables from other subsets are replaced by inputs, which can turns out to be very expensive.

Three-valued logic are often used in model checking partial or approximated systems. For instance, [6] (refined in [7]) used three-valued logic in order to interprete modal logic formulas on partial Kripke structures. However, this work and its refinement ([19], [18], ...) operates on labeled transition systems which are explicitly explored, while our analysis are performed on systems represented as Boolean circuits, symbolically explored using BDD-based techniques. Although applications to symbolic techniques were considered, this has not yet been done to the best of our knowledge.

2 Background

2.1 Finite State Machines

Let $\mathbf{B} = \{0, 1\}$ be the Boolean set. The FSMs we consider are completely specified Mealy machines, defined as tuples $(m, n, p, \delta, \omega, \mathcal{I}, \mathcal{J})$, where:

- m is the number of inputs.
- n is the number of state variables (registers).
- p is the number of outputs.
- $\delta : \mathbf{B}^m \times \mathbf{B}^n \to \mathbf{B}^n$ is the vector of elementary register transition functions.
- $\omega : \mathbf{B}^m \times \mathbf{B}^n \to \mathbf{B}^p$ is the vector of elementary output functions.
- $\mathcal{I} : \mathbf{B}^n \to \mathbf{B}$ is the characteristic function of the set of initial states.
- $\mathcal{J} : \mathbf{B}^m \to \mathbf{B}$ is the characteristic function of the valid input space. For instance, if some inputs are implied by others, or if some pairs of inputs are mutually exclusive, the whole input space would not be valid.

We use the same notation for a set or its characteristic function. Thus, $\mathcal{J}(x) = 1$ means $x \in \mathcal{J}$. Also, for the sake of clarity, we omit the arrow on top of vectorial functions or variables. Negated expressions are either prefixed by $\neg$ or overlined.

2.2 Standard RSS Computation

The usual way to compute the RSS of a FSM symbolically [9, 12], is to find the limit of the converging sequence of finite sets defined by the following equations:

$$\begin{aligned} \mathrm{RSS}_0 &= \mathcal{I} \\ \mathrm{RSS}_{k+1} &= \mathrm{RSS}_k \cup \delta(\mathcal{J}, \mathrm{RSS}_k) \end{aligned} \tag{1}$$

where we use the standard extension of function to sets:

$$\delta(X, Y) = \{\delta(x, y) \mid x \in X, y \in Y\}$$

Using BDDs for characteristic functions, (1) becomes:

$$\mathrm{RSS}_{k+1} = \mathrm{RSS}_k \cup \{\, r' \in \mathbf{B}^n \mid \exists r \in \mathrm{RSS}_k\,,\, \exists i \in \mathbf{B}^m \,.\, \mathcal{J}(i) \wedge r' = \delta(i, r) \,\} \tag{2}$$

In [12], Coudert and Madre introduced the *image* operator $\mathrm{Img}(f, \chi)$, which computes the image of the vectorial function f on the state set of characteristic function χ[1]:

$$\mathrm{Img}(f, \chi) = \lambda r' . \left(\exists r, i \,.\, \chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k = f_k(i, r) \right) \right) \tag{3}$$

[1] $\lambda r'.E$ is the standard λ-calculus notation for the unnamed function of body E, with argument r'.

Algorithm 1 presents an outline of the computation of *RSS* for a given FSM. The main iteration that computes successive RSS_k sets is from line 4 to 17. Line 5 builds the domain for each iteration, based on most recently reached states and the set of valid inputs $\mathcal{J}$. Lines 8 to 10 contain the loop that builds the transition function for the current iteration domain. Line 9 builds the function associated with a single register, restricted for the current domain. Line 10 associates this function with its register variable for the next state and combines it with the final transition function. Line 12 applies the transition function to the last reachable state set. Line 13 performs existential quantifications over the set of old register variables and inputs. Line 14 substitutes the new register variables by the old ones, in order to obtain a function over the set of old register variables for the next iteration. Finally, line 15 computes the sets of new states and line 16 adds this set to the final reachable state set. Iteration stops when the set of new states is empty.

Note that this is only a crude implementation. In the next version, currently under development, the complete transition function is actually never built as we do in lines 8 to 10, which may cause the computation to blow-up quickly. The image is computed over partitions of the transition function and existential quantifications are performed *on-the-fly* rather than in a simple pass, as we mention in line 13. However, it is beyond the scope of this article to discuss such improvements.

```
1   function RSS( FSM )
2     Result ← 𝓘
3     NewStates ← 𝓘
4     repeat
5       Domain ← 𝓙 ∧ NewStates
7       δ ← 1
8       for i ∈ [1..n]
9         δ_i ← BuildRestrictedRegisterFunction( i, Domain )
10        δ ← δ ∧ (NewRegVariable(i) = δ_i)
11      end for
12      Image ← δ ∧ NewStates
13      Image ← Quantify( Image, OldRegVariables+InputVariables )
14      Image ← Substitute( Image, NewRegVariables, OldRegVariables )
15      NewStates ← Image ∧ ¬Result
16      Result ← Result ∨ Image
17    until NewStates = 0
```

Algorithm 1: RSS fixed-point computation

Example Using Algorithm 1, we can enumerate the reachable states of the circuit of Figure 1. The initial state $(1, 0, 0, 0)$ of the circuit is indicated by the values at the bottom of the registers. The first iteration reveals the new

state $(0,1,0,0)$; the second iteration reveals the new state $(0,0,1,0)$; the third iteration reaches the fixpoint: the three registers r_1, r_2 and r_3 are exclusive and r_4 is always 0.

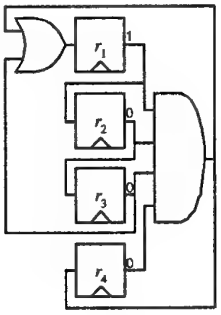


Figure 1: A sequential circuit

RSS Computation Complexity Analysis In this section, the complexity is expressed with respect to the BDD size and in the worst case.

The cost of $\neg$ is constant and the cost of $\vee$, $\wedge$ is polynomial [9, 12]. Unfortunately, the cost of the Img operator, used in line 13 in the Algorithm 1, is exponential with respect to the number of variables, notably because of nested existential quantifications. Informally, while $\exists x \,.\, f(x)$ amounts to computing $f(0)+f(1)$, $\exists x,y \,.\, f(x,y)$ amounts to computing $f(0,0)+f(0,1)+f(1,0)+f(1,1)$, and so on.

In the sequel, we will study techniques to improve the RSS computation by reducing the number of variables to apply *a posteriori* quantification to, in some case at the expense of over-approximation.

3 ORSS Computation

3.1 Replacing State Variables by Inputs

Replacing state variables by inputs can improve the RSS computation: there are fewer register functions to build, combine and manipulate during the image computation, and fewer register variables to substitute. Replacing state variables by inputs weakens the constraints between these variables, leading to an over-approximated result.

Note that the number of *a posteriori* existential quantifications to perform remains the same.

When a state variable is replaced by an input, the correlation between multiple occurrences of this variable in an expression is maintained. This is the case in reconvergent fanout in a circuit. For instance, in Figure 2, there is a circuit fragment generated from a statement like *present I then ... else ...*. The *go* wire, which determines whether a statement is active, is combined with the input I presence wire. Even if the state variable driving this *go* wire is replaced by an input, we are still able to determine that *then* and *else* branches are exclusive.

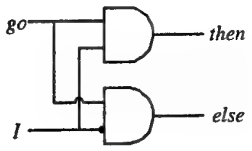


Figure 2: Generated nets for a *present I then ... else ...* statement

Example With the example circuit of Figure 1, suppose we want to check that $r_1 \wedge r_2 = 0$. We can replace r_4 with an input and apply the standard RSS computation algorithm: from the initial state $(1,0,0)$, the first iteration reveals the new state $(0,1,0)$, the second iteration reveals the new state $(0,0,1)$ and the third iteration reaches the fixpoint. We can still prove that $r_1 \wedge r_2 = 0$, but the computation required fewer register functions.

Conversely, if we choose to replace r_3 with an input, starting from the initial state $(r_1, r_2, r_4) = (1,0,0)$, the first iteration reveals the new states $(0,1,0)$ and $(1,1,0)$, the second iteration reveals the new state $(0,0,0)$ and the third iteration reaches the fixpoint. We cannot prove that $r_1 \wedge r_2 = 0$.

3.2 Variable Abstraction using Ternary-Valued Logic

Three-Valued Logic As a refinement of Malik's work [23], Shiple, Berry and Touati [24] used Scott's three-valued logic to analyse cyclic circuits. Scott's three-valued logic is built upon the usual two-valued Boolean logic by adding a third value, noted $\perp$, which means that a variable is *undefined*, and by extending usual Boolean operators.

Similarly, we propose to introduce a third value meaning that a variable is *defined*, i.e. either *true* or *false*, noted d. Indeed, the laws for d are exactly those of $\perp$, and we are simply using standard Scott Logic. However, we prefer to use the d symbol since the intuition is different.

The 3 logic values $\{0,1,d\}$ are respectively encoded by the pairs of Boolean values $\{1,0\}$, $\{0,1\}$ and $\{0,0\}$. In expressions, we encode variables we want to keep by a pair $(x,\overline{x})$, and variables we want to abstract by the constant pair $d = (0,0)$. Three-valued functions (TVFs) are encoded using a pair of Boolean

functions (f^0, f^1), such that f^0 (resp. f^1) is the characteristic function of the set for which f evaluates to 0 (resp. 1). The set f^d of valuations for which f is defined is $f^d = \overline{f^0 + f^1}$ and, by construction, $f^0 \cdot f^1$ is always false. Hence, f does not characterize a partition of two sets $(f, \neg f)$ as in Boolean logic, but a partition of three sets (f^0, f^1, f^d), as seen on Figure 3.

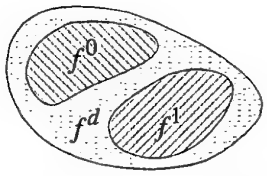


Figure 3: f^0, f^1 and f^d onsets

Standard operators over Boolean functions are extended to TVFs with respect to the following formulas:

$$\begin{aligned} \neg\,(f^0, f^1) &= (f^1, f^0) \\ (f^0, f^1) + (g^0, g^1) &= (f^0 \cdot g^0,\, f^1 + g^1) \\ (f^0, f^1) \cdot (g^0, g^1) &= (f^0 + g^0,\, f^1 \cdot g^1) \end{aligned}$$

For instance, $f+g$ is false if both f and g are false, but true as soon as either f or g is true.

The three-valued logic functions are known to be monotonic [8] in the lattice $\{d \leq 0, d \leq 1\}$.

Application to the RSS Computation By abstracting variables, we take the previous technique a step further: while state variables replaced by inputs were still present in intermediate computation, abstracted variable are completely removed from the support of the BDDs.

To achieve this, we need to return to how the equality in (3) is computed. As the equality $a=b$ can be written as $a \cdot b + \overline{a} \cdot \overline{b}$, (3) is internally expanded into:

$$\mathrm{Img}(f, \chi) = \lambda r' . \left(\exists r, i \,.\, \chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k \cdot f_k(i, r) + \overline{r'_k} \cdot \overline{f_k(i, r)} \right) \right) \qquad (4)$$

Using three-valued logic, we cannot simply replace f_k by f_k^1 and $\overline{f_k}$ by f_k^0, as we do not have $f_k^1 \vee f_k^0$, unlike $f_k \vee \overline{f_k}$, as represented on Figure 3. Instead of a partition f, $\overline{f}$, we now have three sets, f^0, f^1, and $f^d = \overline{f^0 + f^1}$, the latter being the set of arguments for which we only know that f is defined. Therefore, we must *widen* the positive function f by $\overline{f^0}$, and the negative function $\overline{f}$ by $\overline{f^1}$.

We introduce the OImg operator as the widening of the standard Img operator, defined as:

$$\mathrm{OImg}(f,\chi) = \lambda r'.\left(\exists r,i\,.\,\chi(r) \wedge \mathcal{J}(i) \wedge \left(\bigwedge_{k=1}^{n} r'_k \cdot \overline{f_k^0(i,r)} + \overline{r'_k} \cdot \overline{f_k^1(i,r)}\right)\right) \quad (5)$$

Informally, we have replaced the characteristic function of the set "on which f is true" (the *onset* of f), by a superset "on which f is certainly not false", and *vice versa*. However, when applied to concrete variables of the form $(x, \neg x)$, the onsets of f^0 and f^1 forms a partition of the domain on which f is defined, and the result of (5) remains exact.

Since three-valued functions are monotonic, the OImg operator is also monotonic in the complete lattice of sets of states. Hence the algorithm terminates with a unique least fixpoint.

Example Returning to the example circuit of Figure 1, in order to check that $r_1 \wedge r_2 = 0$, we can abstract r_4, and apply the widened RSS computation algorithm: from the initial state $(r_1, r_2, r_3) = (1,0,0)$, three iterations reveal the states $(0,1,0)$ and then $(0,0,1)$. Having abstracted r_4, we could prove that $r_1 \wedge r_2 = 0$ with less functions but also with less intermediate variables.

Conversely, if we choose to abstract r_3, starting from the initial state $(r_1, r_2, r_4) = (1,0,0)$, three iterations reveal the states $(0,1,0)$ and $(1,1,0)$ and then $(0,0,0)$. We cannot prove that $r_1 \wedge r_2 = 0$.

Discussion As for the previous technique, there are fewer register functions to combine and manipulate during the image computation, and even fewer variables to substitute. Furthermore, the number of variables that have to be quantified *a posteriori* is reduced: the former formula $\exists x,y\,.\,f(x,y)$ becomes $f(d,0) + f(d,1)$ when x is abstracted, instead of $f(0,0) + f(0,1) + f(1,0) + f(1,1)$. One can argue that the number of register function to build is increased, but this step of the RSS computation is far from being critical.

On one hand, abstraction reduces the number of BDD variables and functions to compute, by early quantification. Of course, if the abstracted variables were really irrelevant, they would have also disappeared from the BDDs, but *during* its construction; our technique removes them *before*.

On the other hand, we have seen that the equality must be *widened*, which leads to an over-approximated result. Furthermore, the information we loose in the abstraction process is the correlation between positive and negative instances of a variable. For instance, $d\overline{d}$ is abstracted to d instead of 0. Returning to Figure 2, choosing to abstract the *test* variable would lead to loose the knowledge that both *true* and *false* branches are exclusive.

So far, the selection of state variables to be abstracted still depends on proper human designer guidance.

3.3 Refinement Using the Esterel Selection Tree

Esterel [4, 5] is a control-dominated language: the control part has a hierarchical structure, reflecting nesting of statements in the original programs, while communication between different parts of the program is handled through instantaneously propagating signals or shared variables. Roughly, every construct in the program has an associated *selection* wire indicating whether this construct is active or not. The value that these selection wires carry comes from combinations of registers, i.e. the current state of the machine. Selection registers are then combined with tests to activate other areas of the program and finally propagated to the registers to determine the next FSM state.

As generated from high-level language, Esterel circuits feature some interesting information concerning their design, notably the hierarchy of *pauses*, i.e. registers generated by explicit or implicit pause statements. For instance, consider Program 1, where declarations are omitted. Square brackets group statements, semi-colons indicate sequence, and $\|$ indicates parallelism. The *await* instruction contains an implicit pause: once the first instant an *await* statement was activated is over, the next statement is executed as soon as the awaited signal appears. Therefore, each *await* statement will generate a register in the circuit. Because *await* statements at lines 2 and 4 are executed in sequence, their register are *exclusive*; similarly, because block 1-9 is executed in sequence with the *await* statement at line 10, the register coming from line 10 is *exclusive* with all registers coming from block 1-9. On the other hand, blocks 2-5 and 7-8 are *compatible*, so no relation can be infered.

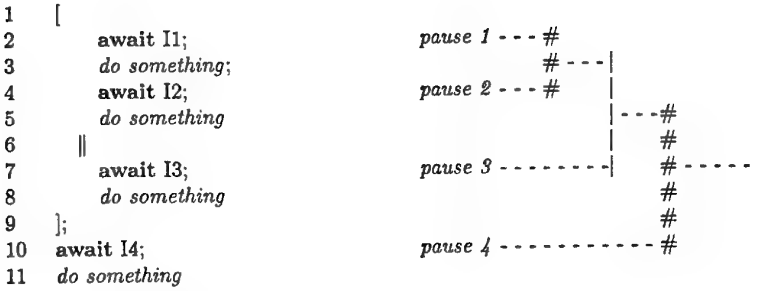
In Esterel circuits, such an information is stored in the *Selection Tree* [25], where non-terminal nodes indicate either compatibility or exclusivity and terminal nodes are registers. The Selection Tree of Program 1 is represented on the right-hand side, where exclusive nodes are noted with sharps. From this Selection Tree, we can build a BDD that alone gives an over-approximation of the RSS of the circuit, for all the states it denies cannot be reached *by construction*. With adequate variable ordering, the construction of such a BDD is straightforward. In the sequel, we will see that this BDD can be used both as an upper bound for over-approximation, and to maintain some constraints on loosen variables.

Use of the Esterel Selection Tree When replacing state variables by inputs, the Esterel selection tree can be used in two ways.

First, we can enhance the input care set with constraints involving at least one state variable that has been replaced by an input, as the input care set can actually reference both inputs and combinational inputs, i.e. real state variables.

Second, we can build, from the relations involving state variables, an upper bound for over-approximation, or *over-approximation ceiling*. Erroneously discovered states that cannot be reached *by construction* are removed by intersecting the set of new states with the over-approximation ceiling, at the end of each step of the RSS computation process.

When abstracting variables, we cannot refine the input care set with relations from the Esterel selection tree, as there is no variable any more. However, we



Program 1: Esterel Example Program

are still able to reference the subset of state variables that are not abstracted, and build an over-approximation ceiling BDD.

4 Experiments

We have implemented the presented technique on top of the TiGeR [13] BDD package. Our tool was run on a 750MHz Pentium III machine with 1GB of memory.

We present data on an industrial Esterel circuit: the fuel-management system of a twin-engine jet aircraft from Dassault Aviation, described in [21]. This system consists in several modules: 2 engines, 2 feeder tanks and several internal and external tanks. The main function of this system is to ensure that the engines are properly fed, while managing system the component failure, the fuel load balancing between the two sides of the aircraft, the in-flight refueling, etc. Most of these tasks are handled by the two feeder tanks, and several safety properties were written for these modules. The complete design has 9,154 nets and 509 registers. Computing the exact RSS of the complete design is intractable on a 1GB machine. However, when focusing on only one safety property at a time, this become largely feasible after a simple pass of transitive network sweeping, which may remove more than 300 registers.

Tables 1 and 2 shows comparisons of the aforementioned approaches to the RSS computation, for two properties of the design that feature regular behavior of our tool (other properties only show behaviours similar to either one of the featured properties). The first line shows the results for exact computation, the second when some state variables are replaced by inputs, the third when some state variables are abstracted, and the fourth when some state variables are abstracted and the Esterel selection tree is used as an over-approximation ceiling. Time and memory columns do not take into account the file parsing and network construction times and memory usages, as they do not depend on the RSS computation approach, and, on such examples, may become the most expensive

part of the process (although their complexity is linear and usually negligible). The ♯L column indicates the number of remaining registers after the transitive network sweeping pass.

Following the advices of the designers, we chose to abstract or replace by inputs the state variables of all of the internal and external tanks but the feeder ones. Note that the hierarchical nature of Syncharts designs allows our tool to work with simple abstraction hints from the designer.

method	time	mem. (MB)	#L	total reachable states at step n				
				1	2	3	4	5
exact	>10mn	79	178	8,749	3.01e8	1.33e13	3.33e13	3.67e13
repl. by inputs	3.8s	6	59	37	341	3,738		
abstracting	1.7s	7	59	37	2.71e5	9.48e6	9.51e6	
abs. + seltree	1.5s	6	59	37	1,670	6,807	7,407	

Table 1: Verification of Property 4

method	time	mem. (MB)	#L	total reachable states at step n						
				1	2	3	4	5	6	7
exact	>2mn	21	120	2	1.66e4	2.41e8	7.03e8	8.85e8		
repl. by inputs	0.6s	5	37	2	70	229	245			
abstracting	0.3s	5	37	2	4.33e3	1.07e6	2.42e6			
abs. + seltree	0.3s	5	37	2	865	7.92e4	1.77e5			

Table 2: Verification of Property 6

By removing state variables, we reduce the number of functions to build and compute the image of. We also reduce the number of existential quantifications to perform. Also, we cut some transitive links between functions, then allowing the transitive network sweeping pass to remove more state variables. When the removed variables are properly chosen, this results in great speed and memory usage improvements: there are several orders of magnitude of differences between the exact RSS computation and the least over-approximation technique. Furthermore, less iterations may be required to reach the fixpoint. Both tables show that abstracting state variables can lead to a greater over-approximation than replacement by inputs; for Property 4, this even require a additional iteration step. However, using relations between state variables expressed by the Esterel selection tree allows us to reduce significantly the over-approximation.

In any case, state variables to be removed must be selected with care. If not, excessive over-approximation may lead to a *snowball effect*: unreachable states are found reachable, then the image of these states must be computed, which

may lead to other unreachable states found as reachable, and so on. As variable abstraction computes greater over-approximations than replacement by inputs, we can naturally expect results to be worse when they are already bad with replacement by inputs.

On another example, time improvements due to variable abstraction range from 20X to 70X and memory reduction from 5X to 10X, but the standard technique of replacing state variables by inputs achieves better results. We are currently improving the ORSS computation algorithm in order to obtain better figures.

5 Conclusions

This paper presents a technique to improve the computation of the Reachable State Space of sequential circuits, by computing over-approximations of it through variable abstraction, using a three-valued logic. This approach takes the commonly used technique of replacing state variables by inputs a step further. When state variables to be removed are properly chosen, relevant improvements of both time and memory usage can be noticed in comparison with replacement by inputs. Excessive over-approximation may be confined by using *by construction* RSS over-approximation ceilings, expressed by high-level structural data, like the Esterel selection tree.

References

1. Charles André. *SyncCharts: A Visual Representation of Reactive Behaviors*, I3S, 1996.
2. Amar Bouali. *XEVE, an Esterel Verification Environment. Proceedings of the 10th International Conference on Computer Aided Verification*, CAV'98, 1998.
3. J. R. Burch, E. M. Clarke, D. L. Dill, K. L. McMillan. *Symbolic Model Checking - 10^{20} States and Beyond. Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, June 1990.
4. Gérard Berry. *The Esterel Language Primer.* CMA, Ecole des Mines de Paris and INRIA. Available with the Esterel system and updated for each release.
5. Gérard Berry. *The Constructive Semantics of Pure Esterel.* CMA, Ecole des Mines de Paris and INRIA. July 2, 1999.
6. G. Bruns, P. Godefroid. *Model Checking Partial State Spaces with 3-Valued Temporal Logics. Proceedings of the 11th Computer Aided Verification International Conference, CAV'99*, 1999.
7. G. Bruns, P. Godefroid. *Generalized Model Checking: Reasoning about Partial State Spaces. Proceedings of the International Conference on Concurrency Theory, Concur'00*, August 2000.
8. J.A. Brzozowski, C.-J. H. Seger. *Asynchronous Circuits.* Springer-Verlag, 1996.
9. Olivier Coudert, Christian Berthet, Jean-Christophe Madre. *Verification of Synchronous Sequential Machines Based on Symbolic Execution. Proceedings of the Workshop on Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Sciences*, June 1989.

An approach to the introduction of formal validation in an asynchronous circuit design flow

Dominique Borrione, Menouer Boubekeur, Emil Dumitrescu (VDS group)
Marc Renaudin, Jean-Baptiste Rigaud, Antoine Sirianni (CIS group)
TIMA Laboratory, Grenoble, France

1. Introduction

Design flows are well established for synchronous designs, and supported with efficient synthesis, simulation and verification software, starting from the Register Transfer level. Asynchronous designs have long been neglected, due to their higher bit cost, and tuning difficulty. However, with the advent of systems on chip involving the interaction between analog and digital parts, and the communication of physically distant circuit modules using long interconnections, the global synchrony hypothesis is no longer realistic; the design of locally synchronous and globally asynchronous systems on chip (SOC's) is a possible answer to the efficient reuse of existing components, using a distributed control and asynchronous interfaces. One of the objectives of the on-going research described in this contribution is to investigate the verification of mixed synchronous and asynchronous circuits, starting from a high-level specification.

Typically, asynchronous circuits are specified at logic level, using a CCS or a CSP-like formalism [Mar90, Ber93]. In our work, we write the initial behavioral description in the CHP language (from Caltech). Following the works of A. Martin [MLM97, MLM99], most of the initial effort at TIMA has been devoted to identifying the conditions required on the initial description, and correctness preserving transformations, in order to synthesize asynchronous circuits. The resulting TAST design flow is shown on Figure 1. Process decomposition and refinements lead to an internal Petri Net formalization, from which a choice of architectural targets is available: micro-pipeline, quasi delay insensitive circuit, or synchronous circuit. A dedicated compiler produces a structural gate network, in source VHDL, for simulation and back-end processing using commercial CAD tools. This approach has supported, up to silicon fabrication, the design of two asynchronous microprocessors: ASPRO[RVR99] and MICA[ABR01].

The recent introduction of formal methods in the design flow has been motivated by an extension of the previous compiler to accept a wider range of CHP primitives, and a need for higher correctness insurance. Up to now, simulation was the only verification means, and it came late in the design process. Trying to model check the synthesis result is difficult because each gate, each flip-flop is a process, each wire is a state element; an event is the arrival of a valid value on a wire, and all inter-leavings of events may a priori occur. Applying brute force model checking quickly leads to combinational explosion.

As an initial feasibility study, we propose to use existing verification tools, even if they were initially not intended for asynchronous designs. More precisely, commercial model checkers used in hardware verification, starting from a standard Verilog or VHDL description, usually assume the existence of a single synchronization master clock for the circuit. In order to provide an early validation of the initial model properties, we extract a state machine from the kernel Petri Net and give a VHDL representation for it. In this first prototype, we introduce a fictitious clock that allows us to examine the model after each transition firing in a sequential path, and to use existing model checking software. In addition, the translation takes advantage of the known restrictions on the Petri Net, being the semantic representation of a CHP program.

In this paper, after discussing the design flow, we describe the main CHP communication and synchronization primitives, their Petri Net model, their "pseudo clocked" VHDL translation, and the

verifications that could be performed on the model, before and after synthesis. We then propose a set of transformations to produce a more compact and efficient coding of the state machine. As a running example, we use a simple, yet characteristic, selector module, and show examples of safety and liveness properties that have been verified using "Formal Check".

2. The TAST Design Flow

At TIMA, we are developing TAST, an open design framework devoted to asynchronous circuits. TAST is the acronym for "Tima Asynchronous Synthesis Tools". It mainly consists in three parts: a compiler, a synthesizer and a simulation-model generator (Figure 1). TAST offers the capability of targeting several outputs from a high level CSP-like description language called CHP (Communicating Hardware Processes).

The compiler translates CHP programs into Petri Nets (PNs) associated to Data Flow Graphs (DFGs). Such a model has been used for years to describe synchronous circuits and systems. However, it finds a particularly adequate application in the field of asynchronous digital circuits and systems design.

The synthesizer is in charge of generating asynchronous circuits from the Petri Net representation of the CHP programs. Asynchronous digital circuit synthesis is based upon DTL (Data Transfer Level) specification. It provides a set of rules to guarantee that PN-DFG graphs are synthesizable into asynchronous digital circuits. TAST synthesizer is so far able to address two kinds of asynchronous circuits: Micropipeline [Sut89] and Quasi Delay Insensitive [Mar90] asynchronous circuits. In the following, only delay insensitive asynchronous circuits are considered.

Behavioral VHDL models of the CHP specification are generated to perform CHP programs verification using simulations (simulation model generator).

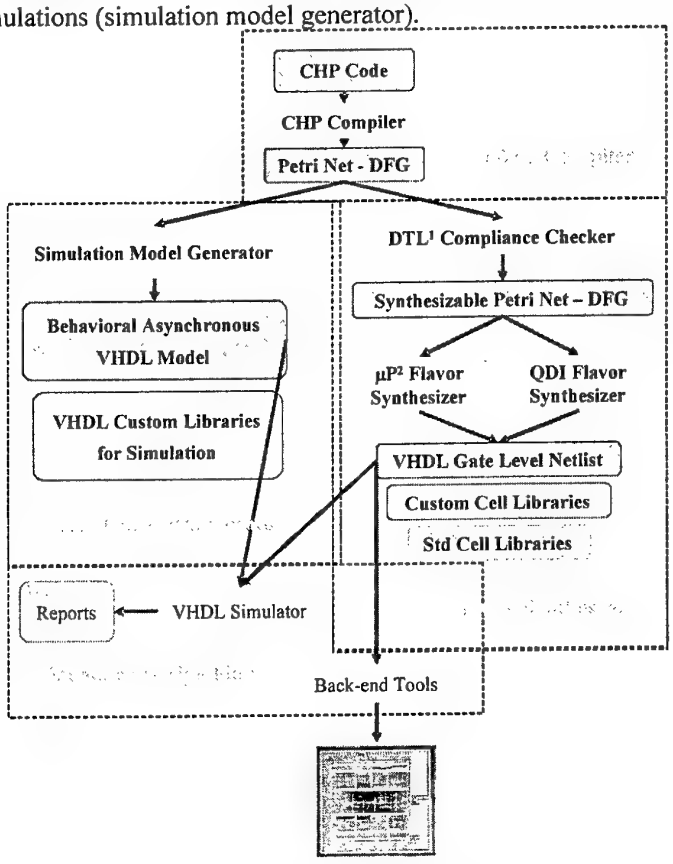


Figure 1: The TAST design flow.

The introduction of a formal verification flow in TAST starts from the Petri Net – DFG format. (Figure 2). A first step consists in choosing an appropriate communication protocol, as well as expanding all communication primitives according to this protocol. A state encoding is associated to the resulting Petri Net, based on its global place marking. From this state encoding, a Finite State Machine interpretation of the Petri Net is constructed and implemented as a VHDL behavioral model. The model obtained can be directly fed to an industrial model-checking tool, accepting a standard HDL entry. The formal verification task consists in modeling the environment of the description we wish to verify, and writing a set of temporal properties that need to be satisfied.
On the other hand, the asynchronous synthesis tool produces a VHDL gate level netlist, which can also be fed to a model-checker. Thus, the compliance of the synthesized model with respect to its specification can be checked, by trying to prove the same set of temporal properties on the specification and the synthesis result.

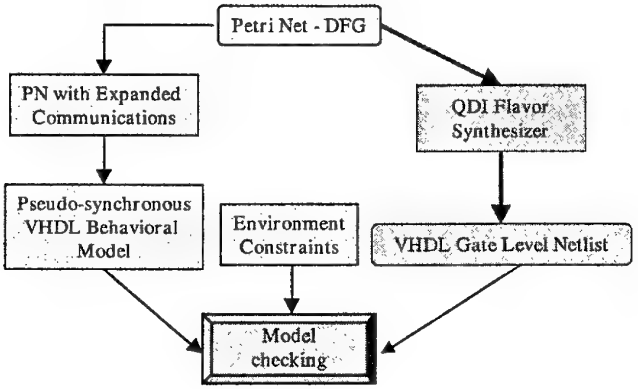


Figure 2: formal verification flow of CHP programs and QDI asynchronous circuits.

3. From CHP programs to Petri Nets

There is no agreement today on a specification language that provides all the facilities to model and synthesize asynchronous circuits. However, CSP-like languages are widely used. Caltech University have proposed CHP (Communicating Hardware Processes) [Mar90][Mar93], Philips has defined Tangram [BKR91][VB93], the university of Utah has developed a tool based on Occam [BS89], and Manchester has defined BALSA [BE97]. All these languages are using the basic concept of CSP : concurrent processes communicating with channels [Hoa78].
We have developed a proprietary high level description language derived from CHP with specific features to cope with communication protocols, data encoding, arbitrary precision arithmetic, non-deterministic data flow, hierarchy, project management and, traceability. All these features make our modified CHP a very practical system description language to develop with.

3.1. CHP syntax basics

In this section, the CHP language is briefly introduced to provide the minimum necessary to enable the reader to read and understand the programs used through out the paper.

Literals
Integers constants are noted in fixed precision with the following syntax:
"<digit>.<digit>..." [base][length] which specifies a vector of "length" element represented in base "base".
Each digit belongs to [0, base-1], and base and length are non-zero natural numbers.
Example : "1.9.7.9"[10] is obviously number 1979.
"1.2.3"[4] = $1*4^2 + 2*4^1 + 3*4^0 = 27$.

In order to avoid this notation for usual bases such as 10 and 2, standard notations are also supported for integers and binary numbers.

Unsigned data types

MR[B] : *Multi-Rail in base B*
This type represents a number between 0 and (B – 1), coded with the "1-of-n" delay insensitive code.
Example :

```
VARIABLE    b : MR [B] ;          -- declaration
            b := "x" [B] ;        -- assignment, with 0 <= x < B
```

MR[B] [L] : *Vector of L Multi-Rails in base B*
This type corresponds to a vector of L elements of type MR [B]. It represents a number between 0 and (B^L - 1).
Example :

```
VARIABLE    b : MR [B] [3] ;      -- declaration
            b := "0.0.0" [B] ;    -- assignment
```

Based on these basic unsigned types, other types are defined to make designers' life easier. They are:

DR : *Dual Rail* – equivalent to MR[2]
DR[L] : *Vector of L Dual Rail elements* – equivalent to MR[2][L]
BIT : *Binary* – equivalent to MR[2]
BIT[L] : *Vector of Bits* – equivalent to MR[2][L]
BOOLEAN : equivalent to MR[2]
NATURAL[Max] : *MR[2][L] type ranging from 0 to Max value.*
L is the superior integer part of (Log_2 [Max+1]). Max is a natural value, which must be specified during the declaration of the element of type NATURAL.
SR : *Single Rail – equivalent to MR [1].*
This type does not carry any value, it allows the specification of synchronizations between communicating processes. The SR type only applies to Channel or Port declaration. It is not relevant for variables.

Signed data types
All the types previously presented, except SR, have a signed equivalent. The complement to the base B is used to encode negative numbers. Thus, the same representation as unsigned types is used, but the highest order digit is interpreted as the sign.
SMR[B] : *Signed Multi Rail in base B*
This type represents a number between -(B / 2) and (B / 2) - 1, if B is even or between -(B - 1) /2 and (B - 1) / 2, if B is odd. The representation of a negative value is obtained with the complement to the base B. For instance in base 16, -2 is represented by 14 (16 + (-2) = 14).
Example :

```
VARIABLE    b : SMR [4];    -- declaration
            b := "2" [4];   -- assignment : b = 2 - 4 = -2
```

SMR[B][L] : *Signed Vector of L Multi Rail in base B*
This type corresponds to a signed vector of L elements of type MR [B]. To evaluate the number, only the highest order digit is signed, the other digits keep their positive values. For instance "1.5"[16] = 1*16 + 5 = 21 while "15.5"[16] = (15 - 16)*16 + 5 = -11.
Example :

```
VARIABLE    b : SMR [3] [3]; -- declaration
            b := "2.2.2" [3]  -- b = (-1)*3^2 + 2*3 + 2 = -1
```

From these basic signed types, other types are defined as abbreviations:
SDR : *Signed Dual Rail* -equivalent to SMR [2].
SDR[L] : *Signed Vector of L Dual Rail values*, equivalent to SMR [2][L].
INTEGER[Max] : *SMR [2][L] type ranging from -(Max+1) to Max.*

Operators

The following operators are available in our CHP language :

Comparison :	=, /=, <, <=, >, >=
Arithmetic :	+, -, *, mod, sll, sla, srl, sra, rol, ror
Logical :	not, nand, and, nor, or, xnor, xor
Communication actions :	! (send), ? (receive), # (probe)
Assignment/conversion :	:=
Sequential/parallel :	; (sequential) and "," (parallel)

Control structures

Deterministic selection. It waits for a unique true guard. Once one of the guards is evaluated to true, it executes the associated bloc and terminates the selection.

```
[   guard1 => bloc1
@   guard2 => bloc2
@   ...
]
```

Non-deterministic selection. It waits for one or more true guards. One of the true guards is selected, and the associated bloc executed. The selection then terminates.

```
[     guard1 => bloc1
@@    guard2 => bloc2
@@    ...
]
```

Deterministic loop. While a unique guard is true, it executes the corresponding bloc. It terminates when none of the guard is true.

```
*[    guard1 => bloc1
@     guard2 => bloc2
@     ...
]
```

Non-deterministic loop. While one or more guards are true, one of the true guards is selected and the corresponding bloc executed. It terminates when none of the guard is true.

```
*[    guard1 => bloc1
@@    guard2 => bloc2
@@    ...
]
```

Program structure

A CHP COMPONENT is made of its communication interface, followed by a declaration part and its body. The communication interface is a directed port list, similar to VHDL. The declaration part declares local objects like channels and constants. The body is made of concurrent processes or component instances. They can all communicate with each other, and also with the component ports. Point to point communication only is allowed.

```
COMPONENT
   component_name
   PORT (port_list)
   {declaration part}
Begin
   component body
End component_name ;
```

A process is made of a port list, a declaration part and a body. It is important to mention that a process is a loop. When the last instruction completes, the process restarts the first instruction.

```
PROCESS process-name
        PORT (port_list)
{declaration part}
[
instruction_list
]
```

Example

As an example, consider the program of Figure 3 which specifies a selector. Because there is only one process, the component port list and the process port list are identical. Channel C is read in the local variable "ctrl" which is tested using a deterministic choice structure. If "ctrl" is '0' then the value read from channel "E" is propagated to channel "S1". If "ctrl" is '1' then the value read from channel "E" is propagated to channel "S2". Finally, if "ctrl" is "3" then the value read from channel "E" is propagated to both channels "S1" and "S2" in parallel.

```
COMPONENT Selector
    PORT ( E: in DR;  C: in MR[3][1];
           S1, S2 : out DR )
Begin
PROCESS main
    PORT (C: in MR[3][1];  E: in DR;
           S1, S2 : out DR )
Variable x : DR;
Variable ctrl : MR[3][1];
[
  *[ C ? ctrl ; [  ctrl = "0"[3] => E ? x; S1 ! x
                @  ctrl = "1"[3] => E ? x; S2 ! x
                @  ctrl = "2"[3] => E ? x; S1 ! x, S2 ! x
             ]
   ]
]
End Selector;
```

Figure 3 : CHP code of a Selector

3.2. Petri Net generation

In order to translate CHP programs into Petri Nets we are considering the Petri Net representations of all the language structures plus those of the sequential and parallel operators. Other instructions and expressions are represented using Data Flow Graphs which are associated to places and transitions. Instructions are associated to places whereas guards are associated to transitions.

Selection operator : [Gi => Ci] with i ∈ {1..n}
The corresponding Petri Net is described in Figure 4. This Petri Net models both types of selection, the deterministic and the non-deterministic one.

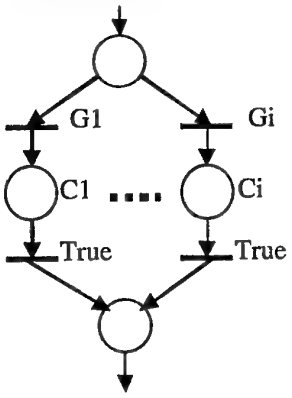


Figure 4 : Petri Net for the selections

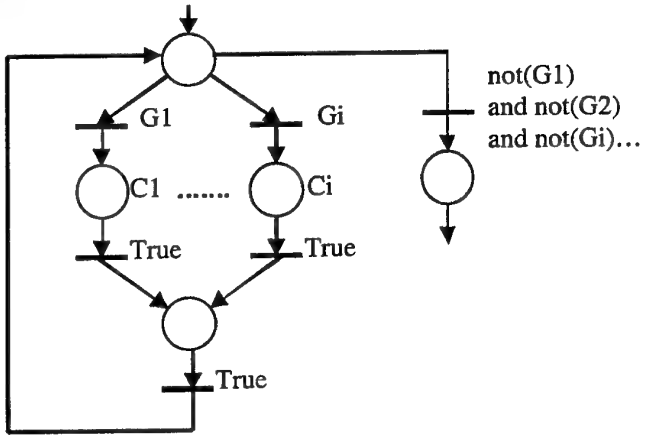


Figure 5 : Petri Net for the repetitions

*Repetition operator : *[Gi => Ci] with i ∈ {1..n}*
The corresponding Petri Net is described in Figure 5. Here again, the Petri Net models both the deterministic and the non-deterministic repetition.

Sequential operator : C1 ; C2
Statement C2 is executed after the completion of instruction C1. The Petri Net of Figure 6 is modelling this operator.

Parallel operator : C1 , C2

Statements C1 and C2 are ex ecuted concurrently. The Petri Net of Figure 7 is modelling this operator.

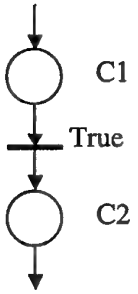


Figure 6 : Petri Net for the sequential operator

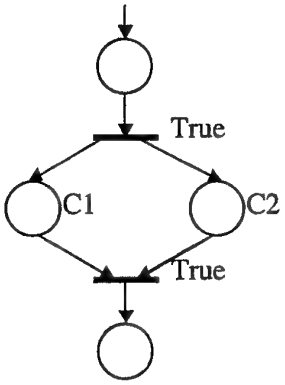


Figure 7 : Petri Net for the parallel operator

Example

Figure 8 gives the Petri Net obtained from the translation of the CHP program of the Selector (Figure 3). Place P0 is the first place of the repetition. This repetition has a unique implicit guarded command which is [True => C ? ctrl...]. Hence, when the command execution completes, the execution has to restart from place P0. Moreover, place Pi is the initial place to start from at the beginning of the execution. Therefore, Pi has to be initially marked.

The first statement to execute is "C ? ctrl". This instruction is represented by a DFG and is associated to place P1. It is followed by a sequential operator, modelled with transition T1. Then the next instruction is a selection starting from place P9. Within the third guarded command note that the parallel operator is represented by places P3, P4 and transitions T3 and T5

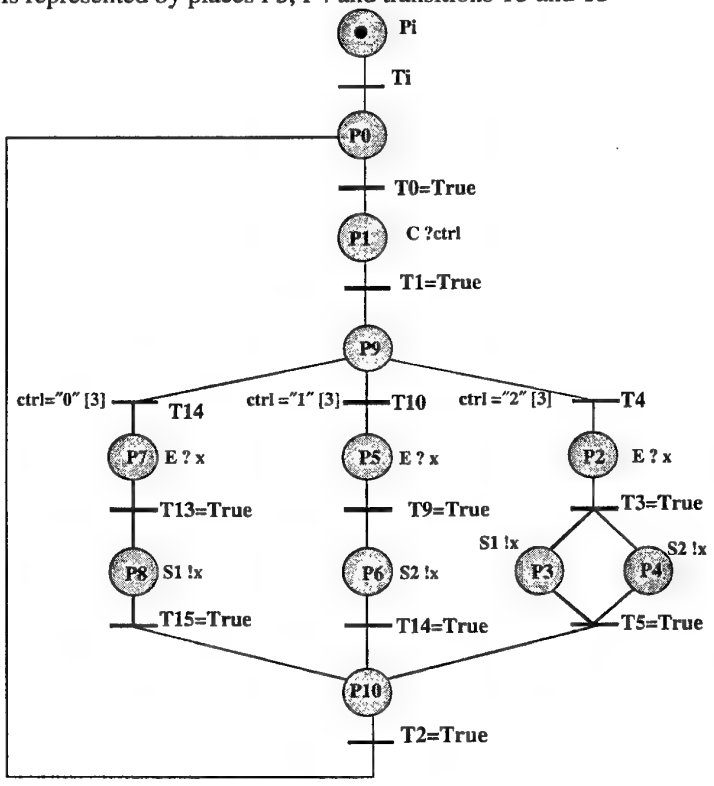


Figure 8 : Petri Net of the Selector.

4. From Petri Nets to gates

First of all, it is worthwhile to mention that deriving Petri Nets from DTL compliant CHP programs ensures that there exists a quasi delay insensitive gate implementation of the Petri Nets. Then, in order to derive a gate implementation from the Petri Net, channel and variable encoding as well as communication protocols have to be precisely defined.

4.1. Data encoding

As discussed in section 3.1, all data are declared and built using the basic MR[B] type. For quasi delay insensitive hardware, type MR[B] must adopt delay insensitive code and implementation. Hence, a digit of type MR[B] is physically implemented with B rails (MR stands for Multi Rail), each wire or rail carrying out one of the B possible values that the digit can take (between 0 and B-1). It is clear that this coding is a one hot coding since at most one bit is one.
Channels are using the same encoding convention. In addition, an acknowledge signal is added to support handshaking communication protocols. As an example the hardware representation of variable "ctrl" and channel E of the CHP program of Figure 3 are depicted in Figure 9.

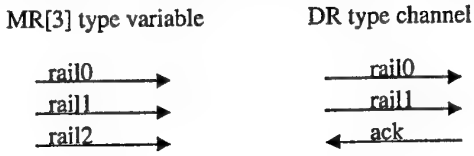


Figure 9 : hardware implementation of variable ctrl : MR[3] and channel E : DR.

4.2. Communication protocols

The 4-phase handshaking protocol is chosen for implementing inter-process communications. Figure 10 illustrates a data transfer along channel E as it is declared in the CHP program of figure 3.

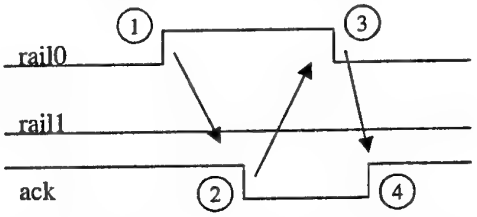


Figure 10 : 4-phase handshaking protocol: transfer of value zero on channel E

Initially, rail0 and rail1 are both zero and the acknowledge signal "ack" is one. In the first phase, the event on "rail0" indicates that a zero is ready in channel "E". In response to this transfer request, phase 2 acknowledges the data by falling down signal "ack". Phase 3 and 4 are necessary to return the signals back to their initial values. A read action from channel "E" using this communication protocol can be formally described using Petri Nets as shown in Figure 11. This representation is commonly called a "handshaking expansion" [Mar90].

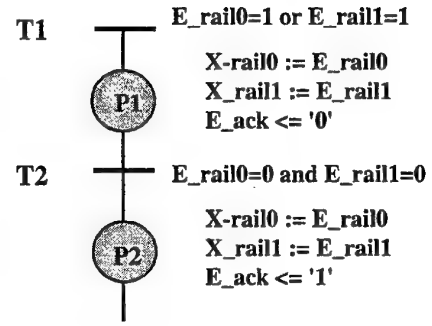


Figure 11 : Petri Net description of a read action on channel "E".
E ? x : with E and x of type DR (Figure 3).

4.3. Synthesis

Generating a gate net-list from the Petri Nets is a difficult problem that is beyond the scope of this paper. The first step of the synthesis process consists in expanding all the communication actions associated to the places of the Petri Net. This is done according to the chosen protocol as illustrated in Figure 11 for a read action. Similar expansions are defined for write actions. Then, all the instructions and guards associated to places and transitions, and represented by DFG constructs, are translated into gates. Finally, the gate net-list is generated from the expanded Petri Net and the synthesized DFG's. The circuit obtained for the Selector is described in Figure 12. Gates denoted "C" are Muller C elements. Gates denoted "Cr" are Muller C elements with reset.

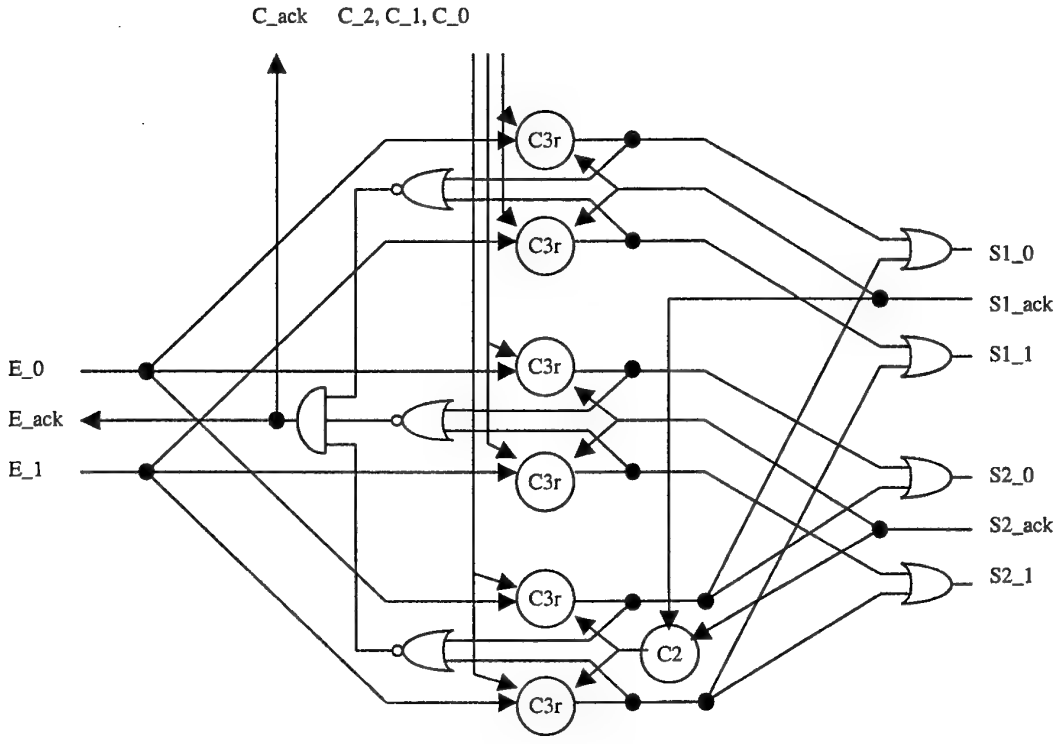


Figure 12 : Gate circuit of the Selector.

5. Verification

Verifying the correctness of the delay-insensitive circuits involves two steps: validation of the initial specification and checking the correctness of the synthesized circuit.
In the first step we validate the initial specification of the asynchronous circuit, by constructing a finite-state machine description in the VHDL language. This FSM corresponds to the Petri net representation. After modeling the circuit environment, the resulting VHDL is checked by applying model-checking tools.
The second step verifies the preservation of the logical properties of the asynchronous circuit after synthesis, with regard to the properties checked on the initial specification.

5.1 Petri nets as finite-states machines

The CHP specification is translated into a *safe* Petri net. We formalize such a Petri net as a quadruple $\Sigma=<S, T, F, M_0>$, where

(i) S and T are finite, disjoint, nonempty sets of places and transitions,
(ii) $F \subseteq (S \times T) \cup (T \times S)$ is a flow relation,
(iii) $M_0 : S \rightarrow$ Bool is the initial marking.

The data-flow graph part of the CHP translation is associated to the Petri net places; in particular, communication actions are linked to places. The internal control flow and the synchronization of the communications are modeled by the guards associated with the Petri net transitions. Each place is represented as a Boolean state-holding object and each marking as a min-term on these objects. This allows a direct translation of the Petri net as a finite-state machine, where the marking represents the global state of the system. Firing the enabled transition t at a marking M produces a new marking M' constructed by setting all input places of t to 0 and all output places to 1. Actions are attached to places and conditions are attached to transitions. Places with communication actions have to be expanded, according to the selected protocol.

5.2 Translation of the Petri net to a verifiable model

In order to benefit from existing industrial tools, and fit in the design flow, we built a prototype which translates the Petri net representation into a register transfer level, behavioral VHDL model suited for formal verification (i.e. compliant to the 1076.6 standard). The simulation-oriented behavioral VHDL model generated in the TAST flow (Figure 1) contains timing and attribute directives which are essential to follow the propagation of the signals, but make this model improper as an input to model checking tools. Ignoring or removing all delays alters the behavior of the model. The trick consists in replacing each unit delay by a tick of a fictitious clock, thus making visible the "delta" delay of a purely behavioral, non-timed, state transition model.

The translation of the CHP COMPONENT interface is straightforward. All the internally declared variables and Petri net places are made VHDL *signals*, which guarantees that all state transitions take at least one step. The overall Petri net for a component is translated as a single process synchronized by the fictitious clock, and not as a set of guarded blocks as in [ABO98]. The full process is executed, and all active transitions are fired, at each step.

a) Algorithm

Input: Petri Net

Translation of declaration part
  Declare the variable according to its type and (or) its parameters (Base and Digit).

Browse the global Petri net (Places and transitions)
IF (place Pi) Then

```
  Associate a signal Pi to the place;
  IF (Place is atomic)
    Add : if Pi then action(Pi) , ... end if ;
              -- action(Pi) : Actions associated to the place Pi.
  else – The place expresses a communication.
    Put_Com () ;
ELSIF  (transition Ti) Then
  Add :
    If (And[cond(Ti), places_in(Ti) ] )
      -- cond(Ti)  Condition associated to the transition Ti.
      places_in(Ti) = false ; -- places_in(Ti) :Input places.
      places_out(Ti) = true ; -- places_out(Ti) :Output places
    end if ;
End IF
End of browsing.
```

Output : VHDL program.

Put_com ()
Implements the communication actions, Read? or Write!, by insertion of the corresponding Petri Net in global Petri Net.

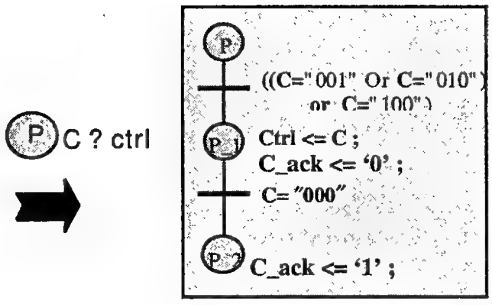
b) Translation of communications

The communication actions are implemented with the handshake protocol.

Translation of Read : (C ?ctrl)

CHP declaration:
C : IN MR[3][1];
variable ctrl: MR[3][1];

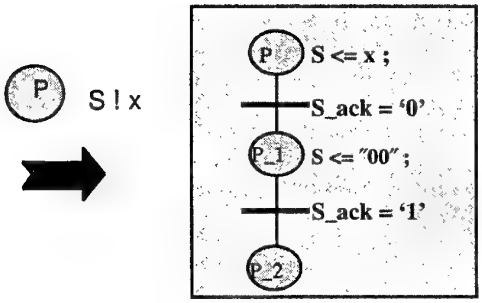
Translation into VHDL:
C : IN bit_vector(2 downto 0);
C_ack : OUT bit; -- Acknowledgement
signal ctrl : bit_vector(2 downto 0);



Translation of : Write (S!x)

CHP declaration:
S1, S2 : OUT DR
variable x: DR;

Translation into VHDL :
S : OUT bit_vector(1 downto 0);
S_ack: IN bit; -- Acknowledgement
signal x : bit_vector(1 downto 0);



c) Application to the selector example

The automatic translation of the selector of Figure 3 reads as follows, where some many similar statements and line-feed characters have been manually deleted for space reasons. Comments are also manually added.

```
entity EX11_Ent is
 port( C : in bit_vector(2 downto 0);
      E : in bit_vector(1 downto 0);
      S1 : out bit_vector(1 downto 0); S2 : out bit_vector(1 downto 0);
      C_a : out bit;   E_a : out bit;
      S1_a : in bit;   S2_a : in bit;
      clk, rst : in bit);
end EX11_Ent;

architecture EX11_a of EX11_Ent is
   signal EX11_MAIN_X : bit_vector(1 downto 0);
   signal EX11_MAIN_CTRL : bit_vector(2 downto 0);
   signal Pi, P0, P9, P7, P8, P1, P5, P6, P2, P3, P4, P0_1, P0_2, P7_1, P7_2,
          P8_1, P8_2, P5_1, P5_2, P6_1, P6_2, P2_1, P2_2, P3_1, P3_2, P4_1, P4_2: boolean;
begin
process(clk, rst)
begin
if (rst='0') then              -- initialization at reset
   S1 <="00"; S2 <="00"; C_a <= '1'; E_a <= '1'; Pi <= true;
elsif clk'event and clk='1' then              -- fictitious clock edge
if P8 then S1 <= EX11_MAIN_X; end if;  -- start of write action at place P8
...          -- same for P6, P3, P4
if P7_1 then EX11_MAIN_X <= E; E_a <= '0'; end if; -- start of read action at place P7
if P7_2 then E_a <= '1'; end if;              -- acknowledge hand-shake at expansion of P7
if ( P7_2) then P8 <= true; P7_2 <= false; end if;  -- Transition T13
if P8_1 then S1 <="00"; end if;
if ((EX11_MAIN_CTRL = "001") and P9) then P7 <= true; P9 <= false; end if;     --OR branch at P9
if ((EX11_MAIN_CTRL = "010") and P9) then P5 <= true; P9 <= false; end if;
if ((EX11_MAIN_CTRL = "100") and P9) then P2 <= true; P9 <= false; end if;
if ((E="00") and P7_1) then P7_2 <= true; P7_1 <= false; end if;
if ((S1_a='0') and P8) then P8_1 <= true; P8 <= false; end if;
if ((S1_a='1') and P8_1) then P8_2 <= true; P8_1 <= false; end if;
... -- same for all other places and transitions
end if;
end process;
end EX11_a;
```

5.3 Verifying the asynchronous synthesis

Along the various synthesis steps, an erroneous procedure can produce an incorrect circuit description. Therefore, it is useful to perform verification after obtaining a synthesized circuit.
The circuit obtained after synthesis cannot be proven equivalent to its specification, because the synthesis process introduces optimization techniques and pipelines; thus the order of actions is not necessarily preserved. As a matter of fact, while the VHDL model for the specification is a single process per CHP process, the synthesized circuit is composed of many concurrent processes. The formal verification can only hope to prove that the essential safety properties that hold for the initial specification are preserved after synthesis.

The same problem that was raised for the specification validation is again encountered: the VHDL simulation model of the synthesized circuits does not conform to the standard verifiable VHDL subset. The asynchronous synthesis tool is based on a library that has to be adapted:

- Muller-C elements contain delays and transparent latches; we changed them into flip-flops, introducing a fictitious clock, as in section 5.2
- All types are transformed into Boolean and Bit.

Figure 13 below gives the transformation of one library component.

```
entity MULLER2_R is
port ( resetb : in STD_ULOGIC;
       S : out STD_ULOGIC;
       A, B : in STD_ULOGIC);
end MULLER2_R;

architecture behaviour of MULLER2_R is
  signal s_s : STD_ULOGIC;
begin

 S<= s_s;

 s_s <= '0' after 1 ns when resetb = '0' else
 '1' after 1 ns when a='1' and B='1' else
 '0' after 1 ns when a='0' and b='0' else
 s_s;

end behaviour
```

```
entity MULLER2_R is
  port ( resetb, clk : in  bit;
         S      : out bit;
         A, B   : in  bit);
end MULLER2_R;

architecture behaviour of MULLER2_R is
signal s_s  : bit;
begin

  S<= s_s;
  process
  begin
    wait until clk ='1';
    if resetb = '0' then s_s <= '0';
    elsif a='1' and B='1' then s_s <= '1';
    elsif a='0' and b='0' then s_s <= '0';
    end if;
  end process;
end behaviour;
```

Figure 13: VHDL models of a Muller-C gate for simulation and for formal verification

5.4 Environment modeling

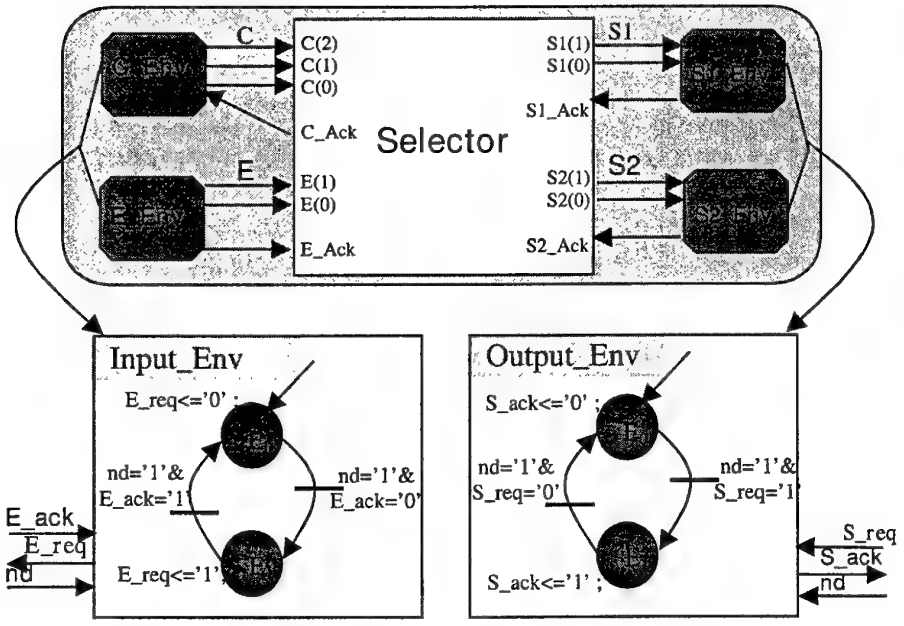


Figure 14: Modeling correctness assumptions on the environment

In order to obtain a correct behavior, a set of environment assumptions must be associated to each communication channel. For a given communication channel, we must ensure that each input wire behavior is compliant to the protocol implemented by the channel. All communication channels behave according to a four-phase communication protocol. Following the direction of the request signal attached to a channel, we distinguish between master channels (the system sends the request) and slave channels (the system receives the request). Thus, a dual behavior must be defined for each

communication channel. For master channels, the dual behavior must receive the request and send a validate within a finite amount of time. Then, it waits until the request falls and deactivates the validate within a finite amount of time. The dual behavior of a slave channel must initiate a transaction, by non-deterministically rising the request signal. The remaining behavior is obtained from the master dual behavior, by exchanging the roles of the request and validate signals.
A dual behavior can be modeled as a non-deterministic description, which is plugged to each channel. A fairness condition must be associated to each dual instance, in order to express the fact that it always reacts within a finite delay.
It is also possible to express a dual behavior as a set of assumptions written in temporal logic.
Figure 14 displays the composition of a system with the dual behaviors attached to each communication channel. Wire nd is a pseudo-input signals used to model non-determinism.

5.5 Application to the selector example

In the case of the selector example, the environment has been described by a set of temporal formulas, and the verification was performed using Formal Check.

Input channel constraint:

C_Env1

After P1 = True
Eventually (C = x "1" or C = x "2" or C = x "4") and P1 = True
Meaning: Each time place P1 is active, an incoming control request will eventually occur.

C_Env2

After C_ack = 0
Eventually C = x "0"
Meaning: Each time the request is acknowledged (C_ack = 0), it will eventually return to zero (C = x "0").

Stable_C

After C = x "1" or C = x "2" or C = x "4"
Always C = stable
Unless C_ack = 0
Meaning: A request is stable until it is acknowledged

Output Channel Constraints

S1_Env1

After S1(1) = 1 or S1(0) = 1
Eventually S1_ack = 0
Meaning: After a request on S (S1(1) = 1 or S1(0) = 1), an acknowledgement will eventually be received (S1_ack = 0).

S1_Env2

After S1 = x"0"
Eventually S1_ack = 1
Meaning: After the write transaction on channel S is finished (S = x "0"), the acknowledgement will eventually be deactivated (S_ack = 1).

The ports (E and S2) constraints are similar to the previous constraints (C and S1).

Some verified properties

The expressed properties correspond generally to the Petri net branches, i.e. the reachability of certain places or Transitions by following a given path. Or they are used to express input-output relationships, as in our example. The characteristic behavior of the selector is described by the three properties A1 to

A3 below. These same properties are verified on the circuit specification and on the synthesized circuit.
Meaning of property A1: If place P01 is active and an incoming request C = x"1" arrives, then a write will eventually occur on S1.

A1) After P01 = True and C = x "1"
Eventually : S1(1) = 1 or S1(0) = 1

A2) After P01 = True and C = x "2"
Eventually : S2(1) = 1 or S2(0) = 1

A3) After : P01 = True and C = x "4"
Eventually : (S1(1) = 1 or S1(0) = 1) and
(S2(1) = 1 or S2(0) = 1)

5.6 Handling combinational explosion

Our asynchronous verification approach applies the symbolic model checking technique to a Finite State Machine interpretation of a CHP program. Obviously, the verification of large descriptions will eventually face combinational explosion. Two main directions need to be explored in order to handle this problem. On the one hand, it is important to ensure that the FSM underlying model associated to a CHP program does not contain irrelevant states. Currently, a FSM state is associated to each possible Petri Net place marking. We are aware that this representation is quite expensive, due to the state encoding which associates one state variable to each place. On the other hand, a verification strategy is required, which exploits the characteristics (symmetry, control and data path, etc.) of particular classes of designs, such as the asynchronous arbiters.

Refining the state machine model
A first improvement to our method consists in associating to the initial PN a FSM model with a more compact state encoding. For instance, states should be represented as values over an enumerated data type. Moreover, the FSM states and transitions should match the PN places and transitions. This correspondence is not always straightforward. "Split – join" (SJ) parallel execution constructions need special processing: generate one separate PN for each parallel branch, as well as the re-synchronization "glue" which marks the end of the parallel execution. Figure 15 illustrates this transformation. The parallel branches are executed as soon as place a is active and transition T1 is true. Each parallel branch makes a signal assignment (S1//S3) and waits for an external event to occur (Req1 = 1//Req2 = 1) before making a second assignment (S2//S4).
By successive application of this transformation, we obtain a collection of concurrent PN's which are free of parallel execution statements. Since only one place at a time may hold value 1 in each concurrent PN, one unsigned signal per concurrent PN can hold the number of the active place.

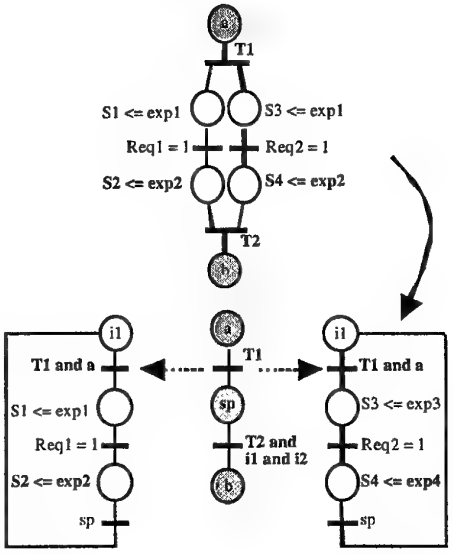


Figure 15 – "split – join" construct transformation

However, further simplifications are still possible on the resulting representation. Petri Nets allow the simultaneous use of two modeling levels in a natural way: sequences of local computations can be mixed with operations that imply *waiting* for an *external event*. Moreover, infinite execution paths that do not contain a wait for an external event are prohibited. Thus, the actual evolution of a PN is only

triggered by external events. Any sequence of local computations performed between two external events can be considered as part of the same place, which only *records their result*. Hence, it is useful to collapse all sequences of places, which only implement local computations. Such sequences are characterized as follows: simple actions like assignments are associated to places; conditions are associated to transitions; one transition may fire as soon as its incoming place is active and its associated condition true. Figures 16a, b and c present a few Petri Net transformation patterns, which perform simplifications according to this criterion. We note them Sa, Sb and Sc. When a transition is a part of a conditional statement, if at least one conditional branch is always true (for instance, they test condition C and not C), one transition occurs immediately (Figure 16a). Parallel and sequential assignments may also be regrouped inside the same place (Figures 16b and c).

The global simplification procedure consists in applying rules Sa and Sc on each appropriate PN construction.

Before applying transformations SJ (Figure 15) or Sb on a parallel construct, a preliminary analysis must determine whether its parallel branches implement any waiting on external events. This analysis performs a recursive coloring of each parallel PN construct:

- each transition is colored if it waits for an external event;
- each parallel structure is colored if its branches contain colored transitions or colored parallel structures.

Transformation SJ must be recursively applied on all colored constructs. Uncolored constructs are treated by transformation Sb.

The FSM model obtained consists of several concurrent state machines that can be represented following a usual coding style. This simplification technique brings two major improvements:

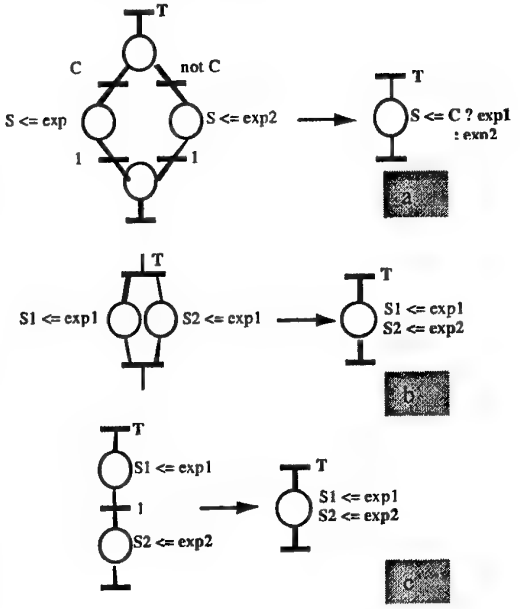


Figure 16 - PN transformation patterns

- The resulting FSM state encoding relies on state enumeration. Our initial implementation associates one state variable to each PN place. For a net containing N places, N state variables are needed. Concurrent FSMs state encoding would only need $\log_2 N$ state variables;
- Irrelevant PN places and transitions (if any) can be suppressed.

Formal verification strategy

The example of Figure 8 implements the selection between incoming channels. Once a channel is selected its data is read and sent through an output channel. This sequence is repeated each time a new command is received. We call it a transaction. Note that a transaction cannot influence the execution of a subsequent transaction. Hence, we may simply focus on writing correctness criteria for a single transaction. According to the value of the control word *ctrl*, the arbiter follows a separate execution path. We distinguish 3 independent execution scenarios (corresponding to ctrl = 0, 1, 2). Hence, each property can be split into three proof sub-goals, one for each execution scenario, following two steps:

- abstracting away the logic driving the variable ctrl, so that it becomes an artificial primary input;
- writing a property for each execution scenario, by constraining ctrl to the corresponding constant value.

If all sub-goals of a property pass, then we may consider that the initial property is true.

This strategy seems adequate for this particular class of designs. It allows important simplifications in the model representation, as each sub-goal constrains ctrl to a constant value, which allows important simplifications of the FSM representation.

6. Related Works

The verification of asynchronous circuits may follow two important directions, according to the asynchronous design approach: untimed (delay insensitive) [Roi97], or timed design. It is well known that the verification of timed systems faces serious complexity problems.
A particular interest has been shown for the development of asynchronous specification methods. The use of concurrent processes for specifying an asynchronous behavior appears to be adequate for most specification problems. Based on this approach, two main directions have been explored: language-based and graph-based asynchronous specification.
Synthesis methods for language-based specifications directly translate a program into a circuit. In [Mar90], the specification program is translated into a circuit by using a series of semantic preserving transformations. Graph-based specifications are used at a conceptual level lower than language based methods. This approach is widely used [Roi97, McM92] together with the Petri net or State Transition Graphs formalisms.
Asynchronous verification methods have been developed following these design approaches. Early works on the formal validation of asynchronous designs include experiments with CIRCAL to model micro-pipe lines, and evaluate correctness and performance properties on them [CM00]. This approach doesn't make use of temporal logic, but models both the system and its properties as processes and constructs the parallel composition of the implementation and property processes. The result is compared to the initial system modeling by using an equivalence checking procedure. If the proof succeeds, the system satisfies the property. This methodology has been used to prove the correctness of two four-phase asynchronous micro-pipelines. [Cla99] presents two semantic models, for the formal verification of reactive systems; they are based on simultaneous or interleaving approaches. It is argued that the interleaving model is more adequate for modeling and verifying asynchronous behaviors. The verification uses special partial order reduction techniques [BN01] in order to handle combinational explosion. In [RC96] a formal verification approach is proposed, in which both circuit specification and circuit environment assumptions are modeled using Petri nets. A state encoding is associated to this representation, which allows the application of BDD symbolic model checking techniques. [YG01] suggests the use of the LOTOS [EV89] specification language together with the CADP [CADP] toolbox for asynchronous verification using model checking.
All these approaches are subject to state space explosion. A number of techniques, such as hierarchical verification [RC95], modular verification [Ha01], abstraction techniques [ZMC99] and Petri net unfolding, already deal with this problem. Like in the RTL systems, the verification of asynchronous circuits heavily relies on the construction of a compact verifiable model as well as on using an adequate verification strategy.

Conclusion

We have implemented an asynchronous circuit design flow based on CHP and VHDL, which includes automatic synthesis, simulation and formal verification. A variety of small circuits (in the category of multiplexors, arbiters, and the like) have been formally verified, by model checking techniques, using pre-existing property checking tools that were initially not intended for asynchronous circuit verification. However, the current prototype FSM generator is not efficient enough to serve the verification of large circuits. Our on-going works concern the development of algorithms for generating more compact state machines, and for finding verification strategies adapted to the known characteristics of various circuit types, as discussed in the section devoted to the handling of combinational explosion. The implementation of these algorithms will be progressively added to the TAST environment for asynchronous circuit designs.

References

[ABO98] R. Airiau, J-M Bergé, V. Olive, J. Rouillard: *VHDL – Langage, modélisation, synthèse*. 2nd edition, Presse Polytechniques et Universitaires Romandes, 1998 (in French).

[ABR01] A. Abrial, J. Bouvier, M. Renaudin, P. Senn and P. Vivet *A New Contactless Smart Card IC using On-Chip Antenna and Asynchronous Microcontroller*. Journal of Solid-State Circuits, vol. 36, 2001, pp. 1101-1107.

[BE97] A. Bardsley, D. Edwards, *Compiling the language Balsa to delay-insensitive hardware*; in C.D. Kloos and E. Cerny, editors, Hardware description languages and their applications (CHDL), pp. 89-91, April, 1997.

[Bel99] Wendy A. Belluomini. *Algorithms for Synthesis and Verification of Timed Circuits and Systems.* PhD thesis, The university of Utah, Utah, 1999.

[BN01] Robert Berks and Radu Negulescu. *Partial-Order Correctness-Preserving of Delay-Insensitive Circuits.* Seventh International Symposium on Asynchronous Circuits and Systems, Salt Lake City, Utah, 2001.

[Ber93] K. Van Berkel, *Handshake Circuits - An Asynchronous Architecture for VLSI Programming.* Cambridge University Press, 1993, ISBN : 0-521-45254-6

[BKR91] K. van Berkel, J. Keyssels, M. Ronken, R. Saeijs and F. Chalij, *The VLSI programming language Tangram and its translation into handshakes circuits.* Proceedings of the European Conference on Design Automation, Amsterdam, pp. 384-389, 1991.

[BS89] E. Brundvand, R. Sproull, *Translating Concurrent Programs into Delay-Insensitive Circuits,* in Proc. ICCAD, pp.262-265, 1989.

[CADP] http://www.inrialpes.fr/vasy/cadp/

[Cla99] Edmund M. Clarke, Orna Grumberg and Doron A. Peled, *Model Checking*, The MIT Press, Cambridge, Massachusetts, 1999.

[CM00] Antonio Cerone, George Milne: *A Methodology for the Formal Analysis of Asynchronous Micropipelines*. Proc. FMCAD 2000, LNCS N° 1954, Springer Verlag, pp.246-262

[EV89] P. H. J. van Eijk, C. A. Vissers, M. Diaz. *The formal description technique LOTOS,* Elsevier Science Publishers B.V., 1989.

[Hoa78] C.A.R. Hoare, *Communicating Sequential Processes.* Communications of the ACM, vol. 8, pp. 666-677, Aug 1978.

[Mar90] A.J. Martin, *Programming in VLSI: From Communicating Processes to Delay-Insensitive Circuits Developments,* in *Concurrency and Communication.* edited by C.A.R. Hoare, Addison Wesley, pp. 1-64, 1990.

[Mar93] A.J. Martin, *Synthesis of Asynchronous VLSI Circuits.* Internal Report, Caltech-CS-TR-93-28, California Institute of Technology, Pasadena, 1993.

[McM92] Kenneh L. McMillan. *Using unfolding to avoid the state explosion problem in the verification of asynchronous circuits.* In GG. V. Proc. International Workshop on computer Aided verification, volume 663, pages 164-177. Spriger-Verlag, 1992.

[MLM97] Alain Martin, Andrew Lines, Rajit Manohar, Mika Nystroem, Paul Penzes, Robert Southworth, Uri Cummings , Tak Kwan Lee: *The Design of an Asynchronous MIPS R3000 Microprocessor.* Proc. 17th Conference on Advanced Research in VLSI, 164-181, IEEE Computer Society Press, 1997.

[MLM99] Rajit Manohar, Tak-Kwan Lee, and Alain J. Martin. *Projection: A Synthesis Technique for Concurrent Systems*. Proc. 5th Int. Symposium on Advanced Research in Asynchronous Circuits and Systems, April 1999.

[Neg98] Radu Negulescu. *Process Spaces and Formal Verification of Asynchronous Circuits.* Universite of Waterloo, Ontario, Canada,1998

[RCP95] Oriol Roig, Jordi Cortadella and Enric Pastor. *Hierarchical Gate-level verification of Speed-Independent Circuits. Polytechnic University of Catalunya*, Barcelona, 1996.

[RCP96] Oriol Roig, Jordi Cortadella and Enric Pastor. *Verification of Asynchronous Circuits by BDD-based Model Checking of Petri Nets.* Polytechnic University of Catalunya, Barcelona, 1996.

[Roi97] Oriol Roig i Mansill. *Formal Verification and Testing of Asynchronous Circuits*, Polytechnic University of Catalunya, Barcelona, 1997.

[RVR99] M. Renaudin, P. Vivet, F. Robin *ASPRO : an Asynchronous 16-Bit RISC Microprocessor with DSP Capabilities.* Proc. ESSCIRC'99, Duisburg, September 21-23, 1999.

[Sut89] I.E. Sutherland, *Micro-pipelines.* Communication of the ACM, Volume 32, N°6, June 1989.

[VB93] K. Van Berkel, *Handshake Circuits - An Asynchronous Architecture for VLSI Programming.* Cambridge University Press, 1993.

[YG2001]M. Yoeli and A. Ginzburg, *LOTOS-based Verification of Asynchronous Circuits*, Technical Report, Dept. of Computer Science, Technion, Hifa, 2001.

[Zh01] Hao Zheng. *Modular Synthesis and Verification of Timed Circuit Using Automatic Abstraction.* PhD thesis, The university of Utah, Utah, 2001.

[ZMC99]Hao Zheng, Eric Mercer, and Chris Meys. *Automatic Abstraction for Verification of Timed Circuits and Systems.* PhD thesis, The university of Utah, Utah, 2001.

Issues in Multiprocessor Memory Consistency Protocol Design and Verification

Ritwik Bhattacharya and Ganesh Gopalakrishnan *
School of Computing, University of Utah
{ritwik | ganesh}@cs.utah.edu

February 25, 2002

Abstract

Distributed shared memory (DSM) systems are central to our advancement in terms of high-performance computing. The complexity of DSM protocol design stems both from the complexity of the high-level notion of correctness—conformance to a modern weak shared memory model—and the degree to which these protocols are aggressive. Many similar issues are faced in the design of related protocols such as in input/output subsystems. In this paper we provide a glimpse of the complexity faced, and our work in progress in addressing these issues through model-checking and synthesis using recently proposed challenge problems to drive our research.

1 Introduction

The computer industry has consistently delivered increasing performance with each new generation of processors. To sustain the current level of growth, *system design and verification complexity* must be kept in check. Design and verification complexity can be minimized through *modular design principles* that allow design and verification results to be *reused*, and through *provably correct automation* that allows verification to be done at *higher levels*. The problem addressed in this paper is how to significantly elevate the level of modularity and automation in the design of *distributed shared memory* (DSM) systems, where the verification complexity has grown out of proportion with raw transistor count.

The DSM discipline continues to be a dominant organizational paradigm for computers. DSM machines may be as simple as a dual-processor desktop computer or as sophisticated as the 512-node ASCI White [1] of Lawrence Livermore. The inherent verification complexity of these DSM machines will be exacerbated by the fact that they will be integrated on single chips. Examples of chip level multiprocessors include the IBM Power4 [2], the Compaq Piranha [10], and the SUN microsystems MAJC 5200 [3]. These chips consist of ensembles of processors, instruction and data level 1 (L1) caches, L2 caches, memory

*This work was supported by National Science Foundation Grants CCR-9987516 and CCR-0081406

controllers, packet switchers, DSM protocol engines, and router hubs. For example, the Piranha has all this functionality available within a *single* 100+ million transistor die. All these processors face a common set of issues when it comes to their DSM protocol design:

- Given the growing computation to communication delays, ownership migrations and invalidations must be handled inexpensively.
- Given the finiteness of resources such as buffer locations, transaction identifiers, etc, deadlock avoidance and buffer reservation schemes must be built into protocols.
- Given the ever widening gap between CPU speeds and memory system speeds, read latencies must be hidden at every possible opportunity. In particular, as many operations must be allowed to happen out of order, implying that *weak memory consistency models* [8] be employed.

The above mentioned problems pertaining to DSM protocol design are, to a large measure, shared by emerging I/O system standards such as 3GIO [4], Infiniband [38], as well as protocols used in networked embedded systems such as Bluetooth [18]. While many implementation methods provide the ability to correct mistakes late in the design cycle (software implementations of DSM protocol engines [11, 23], and software [24] or microcode [31, 10] implementations of hardware DSM protocol engines, for example), the extent of error recovery possible is limited, and the debugging costs are high.

In this paper, we report on our research in progress at the University of Utah which is addressing some of the above issues. The overall goals of our research are as follows. First, we aim to develop techniques that apply to real industrial-scale protocols. Second, we aim to capture recurring design situations as "idioms" and develop reusable verification or synthesis techniques. Last but not least, we aim to have "brute force" verification techniques available to designers so that they can deploy them on new designs that break past patterns, while a formal understanding is being obtained for them. In this paper, we present our specific efforts to achieve the above goals:

- We briefly summarize some of the issues that make DSM protocols complex. To shed more light on these issues, we examine the Wildfire [26] protocol in some detail.
- We survey some of our past efforts in capturing design patterns and setting up a derivational-style synthesis process. We provide a preliminary evaluation of the success of this approach on the Wildfire protocol. This preliminary evaluation reveals how far we can go with respect to an industrial-scale protocol, and what remains to be done.
- We summarize our efforts so far in using "brute force" model-checking, and present our plans to make this process more efficient.

In the remainder of this section, we review some basic terminology and briefly survey related work. In Section 2, we summarize some of the general difficulties of DSM protocol design, and specific issues that come up in the Wildfire protocol. In Section 3, we present the approaches we have tried in the past, as well as plan to try in the near future. Section 4 concludes the paper.

Basic Terminology

DSM protocols have one purpose: manage the *ownership* of cache lines so that maximal concurrency of access is permitted, and the values returned by the reads are according to the desired shared memory model [8]. To understand

shared memory models, consider a simple example. In a multiprocessor, a program `write(A,1); read(B)` running on processor P1 and another program `write(B,2); read(A)` running on processor P2 interleave, producing different execution results for the `read` depending on how *weak* the memory model is: under a strong model such as sequential consistency, one of the `reads` must return[1]either `1` or `2`, while in a weak model such as Total Store Ordering [39], both the reads can return `0`. To drive home the connection between memory models and synchronization code, consider one simple example, namely Peterson's algorithm for mutual exclusion [35]. This mutual exclusion protocol fails to work under TSO, but works under sequential consistency.

In general, modern weak shared memory models are far more intricate than either sequential consistency or cache coherence. They support ordinary- as well as special reads and writes that obey different ordering properties. Furthermore, these orderings depend on which address locations - coherent, non-coherent, or I/O - these reads and writes fall on. While the weak ordering rules impose a burden on writers of synchronization libraries, they provide considerably more opportunities for compiler writers and hardware designers to gain performance through re-orderings.

The *correctness problem* that we allude to in this paper is one of showing that all executions generated by a DSM multiprocessor for any given concurrent program are also allowed by the shared memory consistency model. This goes far beyond the scope of what is traditionally known as "*cache coherence verification*" where the correctness is only with respect to a *single* address appearing coherent (consistent) for all processors. In this sense, both sequential consistency and TSO obey *coherence*; however, as the Peterson protocol example shows, they are not equivalent shared memory consistency protocols.

Related Work

The area of shared memory consistency models is vast. We do not attempt a survey; for details, please see [8, 5]. In [19], Grahn studied many contemporary DSM protocols in a unified setting. He also compared the implementation details of achieving synchronization. The use of formal operational models of memory consistency to verify assembly code synchronization routines is studied in [16]. In [34], several protocols, including Stanford FLASH [24, 20] are shown to be sequentially consistent, using theorem proving. Additional related works in specification and verification of DSM protocols include [29, 5, 14, 36, 7, 9].

2 Complexity of DSM protocols

2.1 General issues contributing to complexity

To provide a concrete context for our discussions, we now take-up one commonly occurring design scenario from DSM protocol design: the 'three-way handoff' scenario of handling a missed write at a processor, say P1. In this scenario, processor P1 requests the directory controller of a cache line for an exclusive copy of the cache line before it can proceed with the write. The directory controller, in turn, requests the current owner (another processor, say P2) of the line to forward the line directly to P1. Completely unawares, P2 may have already decided to evict the line, which is in flight in the form of a *line relinquish*

[1]We assume an initial memory value of `0` at all locations.

message towards the directory. In this case, the directory controller receives an "unexpected" line relinquish from P2 when it was expecting a "forwarding to P1 done" acknowledgement. In most designs, the directory controller would recover from such a state by giving priority to the cache controller's attempt over its own attempt. Consequently the directory controller acts as if it has been *implicitly nacked*[2]. It collects P2's relinquished line and hands it over to P1. Over and above these correctness considerations, one must also take into account the other crucial factors, such as the following:

- *Message ordering properties of the interconnect medium:* usually several *priority lanes* are employed to allow messages to re-order, partly to improve performance, and partly to avoid deadlocks. Each message above must be sent on the 'correct' priority lane.
- *Buffer capacities:* Usually, before sending a request, a requester, X, must reserve a spare location in its input queue to be able to receive an acknowledgement from its requestee, Y. However, sometimes a single spare location may not suffice, as the following scenario of 'tail of buffer livelock' illustrates. In this scenario, when the acknowledgement is outstanding, another requester, Z, may send a request to X. This may happen precisely when Y is about to respond. Since the only available buffer slot is occupied by Z's request, X is forced to send back Y's acknowledgement. It then examines Z's request which, it usually cannot process, as X is "in the middle of a request itself." X, thereby, ends up *nack*ing Z's request also. This scenario can repeat indefinitely.
- *Respect the memory ordering semantics:* The above protocol actions describe how *coherence* - strong ordering with respect to a *single* cache-line - is attempted to be maintained. It is unclear whether the ordering constraints for *multiple addresses* as dictated by *sequential consistency* or *weaker memory models* are being met.

The Wildfire protocol involves virtually all the above issues plus some, as described in the next section.

2.2 Additional complexities of the Wildfire protocol

We first describe briefly the Alpha memory model and the Wildfire cache coherence protocol, which we are using as a driving example in our research. We then give some example scenarios that show the complexity of the protocol.

2.2.1 The Alpha Memory Model

In the Alpha memory model we consider [26], a processor can issue five different kinds of memory requests, which are named Rd, Wr, LL, SC and MB. A Rd is a read of a memory address, and the memory responds with a data value. A Wr is a write to a memory address. The memory responds with an acknowledgement that the write has been performed. An LL is a 'Load Locked.' This is a read of a memory address that 'locks' the address for the processor, unlocking any other address that might be locked for that processor. The memory responds with a data value. An SC is a 'Store Conditional.' This is a conditional write of a memory address which succeeds if the specified address is locked, and fails otherwise. If it fails, it does not update memory. An SC always (successful or

[2] *nack* stands for negative acknowledgement.

not) unlocks any address that is locked by the processor. A memory address is also unlocked when a different processor performs a Wr or a successful SC to an address locked by the processor. The memory responds to an SC with an acknowledgement indicating whether the request succeeded or not. An MB is a 'Memory Barrier.' Two requests from the same processor to the same address must always be performed in the order in which they were requested. Two requests to different addresses from the same processor must be performed in the order they were requested if an MB request was issued after the first and before the second. The memory does not respond to an MB. The *source* of a memory operation is the most recent write to the memory address involved in the operation (see ordering rules below).

The orderings that the Alpha memory model imposes on the above five operations are as follows:

- A request r_1 precedes another request r_2 if r_1 is the *source* of r_2.
- r_1 precedes r_2 if r_1 and r_2 are requests from the same processor, r_1 precedes r_2 in the request queue, and either at least one of them is an MB, or they are requests to the same address.
- Writes and successful SCs to the same location that have issued responses are totally ordered.
- LL's and successful SC's are properly paired, without intervening writes by another processor to the same address.

2.2.2 The Wildfire Protocol

The Wildfire cache coherence protocol was designed to implement the Alpha memory model. It models a NUMA shared memory system, and consists of a network of processors and memory connected to local switches, which are all in turn connected to a single global switch. An example is shown in Fig 1. The global space of memory addresses is partitioned among the local switches, with each address belonging to exactly one local switch, which is the home node for that address. Each processor has a cache that contains local copies of some addresses. Wildfire is a directory based cache coherence protocol where each local switch maintains a directory with entries for each address that belongs to it. Each entry has information on which processors currently have a copy of that address in their local cache.

Processors communicate with each other and with local switches through messages. If a processor needs to send a message to a local switch other than the one it is attached to, it sends the message to its local switch, which relays it to the global switch, which in turn relays it to the correct local switch.

Processors are connected to their local switch by two unidirectional queues. The same is true for local switches and the global switch. Messages in the queues are allowed to reorder, subject to certain constraints. All request and control messages travel along this set of queues. There is also a separate network of queues that connect processors directly to one another. These queues are used only to send actual data values by the owner of an address.

An entry in the processor's cache is in one of the following states :

- *Exclusive*: This is the primary copy of the data. The processor can read or write it.
- *SharedClean*: The copy is a secondary, read-only copy.
- *SharedDirty*: This is the primary copy of the data, but it is read-only (The state was Exclusive, and then some other processor requested a read-only copy).

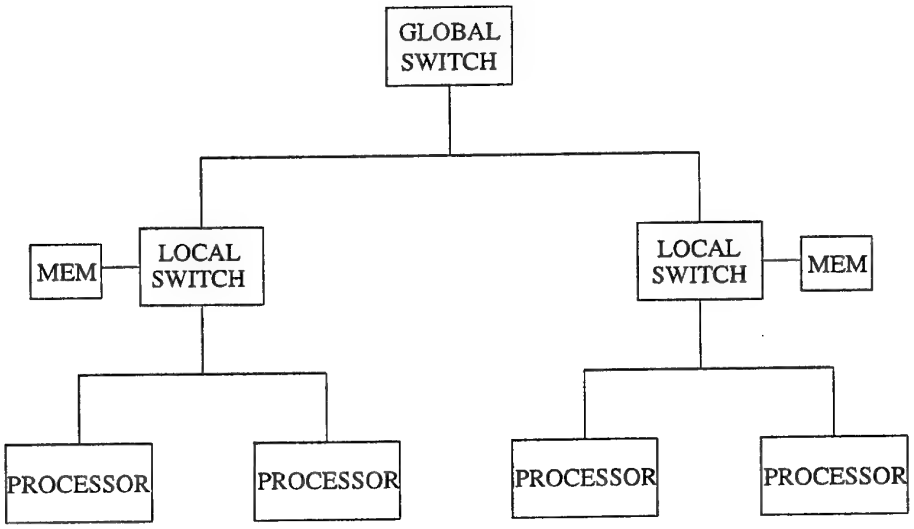


Figure 1: Example of a Wildfire multiprocessor system

- *Invalid*: The cache does not contain a copy of this address.

The state of a cache line changes as a result of the various requests that processors issue, and the responses to those requests. As mentioned earlier, these communications are carried out through messages. An indication of the complexity of the protocol is that there are 13 different *types* of messages in the protocol, and there can of course be multiple messages of each type at any given time in the system. Completing a single memory operation (such as a Wr or a Rd) can require exchanging as many as seven messages among three different components of the system, all of which can be interleaved and reordered with other messages in the system. There are many other subtleties about the protocol that lead to its complexity, but are beyond the scope of this paper. Please see the full Wildfire documentation at [26] for more information.

2.2.3 Scenarios of the Wildfire protocol

We now discuss some scenarios that illustrate the complexity of the Wildfire protocol. First, we describe how Wildfire handles the situation described in Section 2.1. A processor maintains multiple *versions* of the same cache line, each with a possibly different data value. Thus, when a processor evicts a cache line, it does not delete the version of the cache entry until an *ack* for the eviction has been received. So when its evict message crosses over with the request from the directory to forward the line, it can handle the forwarding request, since it has kept the version around. This data is then directly sent to the requesting processor via a separate 'fill channel.'

Our second scenario describes what is known as the shadow mode in the Wildfire protocol. For this scenario, suppose that processors p_1 and p_3 and address a_1 are on local switch ls_1, and processors p_2 and p_4, and address a_2 are on local switch ls_2. Also, assume that p_1 has the primary copy of a_2 in it's cache, and similarly p_2 has the primary copy of a_1. Now, the following events occur [26]:

– p_1 requests a copy of a_1.

- p_1's request reaches ls_1, which sends a request towards p_2 to forward the data to p_1, and updates it's directory entry for a_1 to record p_1 as the new owner.
- p_2 requests a copy of a_2.
- p_2's request reaches ls_2, which sends a request towards p_1 to forward the data to p_2, and updates it's directory entry for a_2 to record p_2 as the new owner.
- p_3 requests a copy of a_1.
- p_3's request reaches ls_1, which directly puts a request to forward the data in p_1's queue.
- p_4 requests a copy of a_2.
- p_4's request reaches ls_2, which directly puts a request to forward the data in p_2's queue.
- Lots of other unrelated messages enter the queues for p_1 and p_2
- The forward request for p_1 to send the line a_2 to p_2 reaches ls_1
- The forward request for p_2 to send the line a_1 to p_1 reaches ls_2

Now, the request for p_1 to forward the line a_1 to p_3 is at the head of p_1's queue, but it cannot handle it yet, since it does not have the line a_1 (though the directory ls_1 thinks it does). Similarly, the request for p_2 to forward the line a_2 to p_4 is at the head of its queue, but p_2 cannot handle it either, since it does not have the line a_2. The request for p_2 to forward the line a_1 to p_1 is stuck at ls_2, since p_2's queue is full with the other unrelated messages. Similarly, the request for p_1 to forward the line a_2 to p_2 is stuck at ls_1, since p_1's queue is full with the other unrelated messages too. We have now reached a deadlock, essentially due to the fact that p_3's (p_4's) request for line a_1 (a_2) gets directly inserted into p_1's (p_2's) queue.

This situation is avoided in Wildfire by the use of 'shadowing'. What this says is that whenever an address that is owned by a directory is in the cache of a processor not directly connected to that directory (local switch), *all* messages related to that address must go through the global switch, even if both the source and destination of a message are on the same local switch. In the above situation, this condition will result in p_3's (p_4's) request for line a_1 (a_2) to be routed through the global switch, ensuring that either p_3's request for a_1 will end up behind p_2's request for a_2 in p_1's queue, or p_4's request for a_2 will end up behind p_1's request for a_1 in p_2's queue, thus eliminating the deadlock.

It is nontrivial to prove that the above solution works in general, and actually involves proving that Wildfire satisfies the liveness property of the Alpha memory model.

Hopefully, the above scenarios will serve to convince the reader that today's memory consistency protocols are highly complex systems, and hence verifying their correctness is commensurately hard.

3 Verification Effort

In this section, we describe our efforts at verifying the Wildfire protocol. We discuss our attempts at using the TLC model-checker[25] (3.1), the Murφ system [15] (3.2), and an experimental synthesis technique [30] (3.3).

3.1 Verification using TLC

Both the Alpha memory model specification and the Wildfire protocol are written in TLA[27], which is a logic for specifying and reasoning about concurrent systems. Hence, our initial attempt to verify the Wildfire protocol focussed on using TLC, which is an explicit state enumeration based model checker for specifications written in TLA. We soon realized, however, that TLC would not be adequate, for two reasons. For technical reasons that are beyond the scope of this paper, TLC cannot handle specifications that are not *machine closed*, which is the case with the Alpha specification. A specification is said to be machine closed if it's liveness property does not introduce additional safety properties into the system [6]. The second, practical reason is that TLC is written in Java, making it extremely slow. The Wildfire model was allowed to run for seven days, and TLC had only explored around 15 million states, for a rate of around 1400 states/minute. This was unacceptable for any practical verification to be done. The TLC verification run did find a 'bug', but that turned out to be a bug in TLC itself, and not the protocol. This was reported to the authors of TLC, who have said that TLC will be fixed to remove this bug. This bug has to do with the way TLC handles conjunctions.

3.2 Verification using Murφ

We next decided to code up the Wildfire protocol in a different modeling language, so we could carry out the verification with another tool. We debated on whether to use a symbolic state representation based tool such as SMV [28], or an explicit state enumeration based tool like Murφ or Spin. SMV was ruled out as its input language is too low level for coding the protocol at the required level of abstraction. Also, based on the experience of Hu [22], it seemed likely that there would be a considerable blow-up in the BDD size. In the end, we decided to go with Murφ, as it has a reasonably expressive input language, and is also quite efficient. The Murφ coding effort took about a month. During this time [37] adapted an existing parallel version of Murφ to run using MPI libraries, so it could be run on a network testbed at the University of Utah which provides up to 172 networked workstations to run experiments on. The sequential version of Murφ soon consumed all the memory resources of the largest single machine at our disposal, which is an SGI with 4 GB of RAM. We then started a run of the parallel version on the network testbed. Each machine in the testbed has 512 MB of RAM. Our initial experiment with 15 such nodes explored about 3.3 million states in 4 minutes before crashing the system. This crash is thought to be due to a bug either in the MPI libraries, or the port of Murφ to MPI, and is being investigated. But the important lesson we learned is that even for the small instantiation of the problem we chose, with two processors, one address, and queue sizes of three, the theoretical total state space is around 48 million (using hashing of the state vector to 40 bits), suggesting that for a realistic instantiation, the state space might be beyond the capabilities of conventional model checking. This has lead us to explore a completely different technique for deriving such protocols, which we discuss in the next section.

3.3 DSM Protocol Synthesis

There have been very few attempts at developing formal approaches to DSM protocol design that take higher level protocol specifications and generate effi-

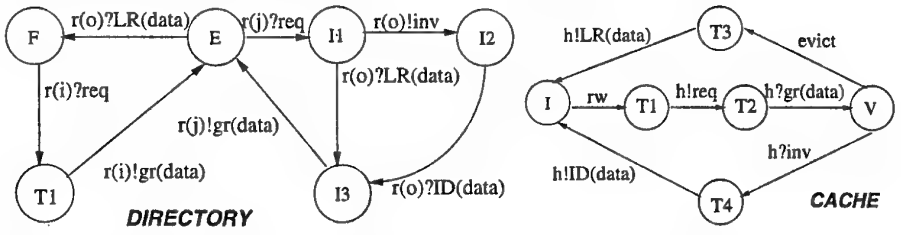


Figure 2: Directory and Cache controller Protocols of Migratory Scheme

cient protocols. We call the higher level protocols *atomic transaction* protocols and the lower level protocols *split transaction* protocols. In all the approaches we have seen, it is assumed that protocol designers must take charge of creating the split transaction protocols manually for obtaining maximally efficient implementations. However, this approach is highly error-prone. The main difficulty is the *huge* semantic gap between shared memory models and split transaction protocols. Protocol synthesis is feasible because the techniques employed by designers for handling ownership transfers, performing invalidations, and speculative protocol execution in real protocols tend to recur quite frequently. Protocol synthesis will allow verification to be employed at a higher level of protocol representation. This, in addition to reducing the state space, will render DSM protocol behavior *semantically closer to human intuition*, unlike in today's approaches where the low level state-space mixes the important (e.g., synchronization completing) with the mundane (how one message overtakes another inside a buffer).

Previous work in our group [30] has demonstrated the feasibility of protocol refinement. We defined a formal refinement relation, and showed, using the theorem prover PVS [32], that the protocols synthesized by our rules stand in this relation with respect to the atomic transaction protocols. It was shown there that the atomic transaction protocol for a two processor Avalanche system [13] had only 54 states, whereas the split transaction protocol had 23,000 states. Since our aim is to do the verification on the atomic transaction protocol, this has the potential to significantly reduce the state space. We briefly summarize our past work in the next section.

3.3.1 A brief overview of protocol synthesis through refinement

We adopt the CSP notation of Hoare [21] and model ownership transfers as *atomic* transaction steps (using the rendezvous construct of CSP). In Figure 2, we illustrate such a protocol description. In this description, the ownership of a cache line is managed under a *migratory-style* protocol: the line access rights migrate from node to node. As a specific example, assume that there are two cache controllers C1 and C2 executing a cache node protocol beginning in state `I` (invalid), and one directory controller D executing the directory protocol beginning in state `F` (the line is 'free'). When C1 suffers a cache miss (the `rw` transition), it attains state `T1`, and sends a request to the directory controller. We show the act of sending this request through the rendezvous statement `h!req`. This statement is jointly executed with the matching statement `r(i)?req` of D. Note that we do not mention *how* this rendezvous is realized. All that the specification writer needs to think about is that the state of D and C1 (viewed as a pair) advances from `(F,T1)` to `(T1,T2)` *atomically*. D then executes the `r(j)!gr(data)`, granting the data and line ownership to

C1. This again causes an atomic advancement of the state from (T1,T2) to (E,V). Suppose C2 now suffers a miss, attains state T1, and sends a request to D. The joint moves of D and C2 will now be from (E,T1) to (I1,T2), the state where D is preparing to invalidate C1 (which is, recall, in state V). Then, D and C1 execute the joint move (I1,V) to (I2,T4) to (I3,I), completing the invalidation of C1. Then, D and C2 execute the joint move (I3,T2) to (E,V), where D now records the ownership of C2.

Notice that while D and C2 were in state (I1,T2), C1 could have decided to autonomously evict the cache line, attaining state T3. Unfortunately, if D now initiates the rendezvous r(o)!inv, instead of a matching action occurring, what occurs is a rendezvous initiation by C1 for another statement, namely h!LR(data). This is quite akin to situations that have been faced by past researchers when they studied the problem of implementing CSP on a distributed platform using *generalized input- and output guards* [12]. Unfortunately, their solutions are too "heavy weight" for our purposes. Our solution to handle the above "apparent deadlock" situation is to somehow make D aware that it has been *nack*ed (in our static priority scheme, it is D that always backs-off in such situations). We then make D bounce back to state I1, and participate in the LR rendezvous which can now succeed.

Synthesis into split transactions

Split transactions (handshakes) consist of asynchronous (non-blocking) *send*s and receives. A rendezvous construct h!req is initiated through an asynchronous send, denoted h!!req. The handshake completes when a reply message h??req is received back. For simplicity, in our past work we disallowed a cache controller process from having a generalized guard (input and output actions present). In fact, if a cache controller process has an active rendezvous (of the kind 'p!E'), then it must be the only guard in a communication state[3].

Thus, the rendezvous of a cache controller must not fail - while that of the directory controller can be *nack*ed. (This decision stems from what is really practical: a directory controller can be *nack*ed; however, if a CPU misses on a cache line, and its cache controller sends a request for that line, there is no option but to supply that line.) We also imposed the discipline that a cache controller may not communicate directly with another cache controller. While this decision simplified our algorithm, it prevented certain advanced types of refinements to be elaborated shortly. We will revise these decisions in our future work.

A brief overview of our synthesis procedure is as follows. The cache controller is either in an 'internal' (non-communication) state or a communication state [4]. Assume that it is in the latter. It must then reserve a buffer location to receive a reply, and then send out a request for rendezvous. If this request is *nack*ed (say, because of buffer capacity running out at the directory controller), it must retry its request. The directory controller, on the other hand, must reserve *two* free locations before it initiates a rendezvous. The extra location is to prevent the 'tail of buffer livelock' scenario illustrated earlier. It must try to engage in one of the rendezvous allowed by its communication state. Unlike a cache controller, a directory controller can be written with generalized guards. It can try these guards in turn. Figure 3 illustrates the *refined* (split

[3] A communication state is one where one of the guards is a communication action.

[4] Notice that as in CSP 1978, we named processes directly and did not use channel names. We may revise this decision in the present work.

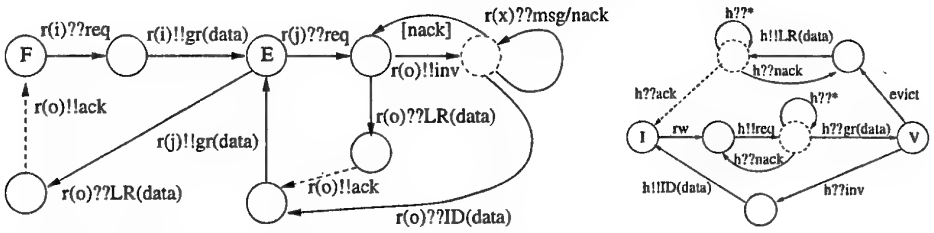


Figure 3: Refined Directory and Cache controller Migratory Protocols

transaction) protocols of the directory and cache controllers. While space precludes a full explanation, the asynchronous handshaking and details of **nack** handling are highlighted by this figure. We argued, using paper-and-pencil, that our algorithm ensures forward progress. Our algorithm can also elegantly handle the earlier mentioned 'unexpected' situation' of P2 relinquishing a line concurrently with the directory controller requesting it.

We showed, using the PVS theorem prover [33], that the split transaction protocol provides all (and only) those executions that are allowed by the atomic transaction protocol. In other words, the sequence of completed rendezvous in the implementation is one of the ones allowed by the specification, and furthermore, all the ones in the specification are realized by the implementation. *Thus, if a designer writes code to performs cache- and directory update activities around such as 'synchronization skeleton' at the atomic level,* the same activities will manifest at the split transaction level. The biggest advantage of the split transaction level is, of course, that it is *implementable*! The atomic transaction level is easier to understand and verify, but cannot be implemented directly.

An Assessment

We applied the technique described in [30] by hand to the Wildfire protocol. We found that if we had the following three synthesis rules in our algorithm, we could have synthesized all high-level aspects of the Wildfire protocol:

- The three-way handoff scenario, as described above in Section 2.1.
- A scenario that stems from three-way handoff. In this scenario, P2 must keep a copy of the cache line around, should the directory be allowed to sending "forward" requests from other nodes towards P2. This capability is, actually, necessary whenever three-way handoff is used because of the arbitrary delay between when P2 relinquishes ownership to when the directory becomes aware of that.
- A still more intricate scenario found in Wildfire. This scenario must be supported only if the designer is (as in Wildfire) extremely aggressive, and allows P2 to re-acquire as well as relinquish the very same cache line multiple times, while it is waiting for the directory to receive and acknowledge its very first relinquish! (In Wildfire, this forces P2 to keep multiple versions of the line - "one for itself" and the others for "forward requests".)

These results have encouraged us to pursue the goal of protocol synthesis further. We discuss our plans for enhancing current techniques, and other future work, in the next section.

3.4 Future Work

We plan to attack the problem of verifying the correctness of DSM protocols from two different directions. One approach will be to try and develop better techniques for verifying existing split transaction protocols. Towards this end, we are in the process of developing a partial order reduction algorithm to be implemented in Murφ. This will be a reformulation of the classical partial order reduction algorithm [17], without the notion of processes. All implementations we have seen use heuristics based on partitioning the transition relation based on processes. We plan to use a different technique to partition the transition relation. The advantage is that this technique can then be used on systems that are modeled directly as transition systems, without any clear distinction of processes. We also believe that in some cases, this may result in more partial order reductions than in the classical algorithm.

Our other angle of attack is to further develop the idea of DSM protocol synthesis, by adding rules to the synthesis algorithm that exploit re-orderings permitted by the underlying memory model. This will result in more efficient synthesized protocols, that will be competitive with hand-designed protocols. Even if the synthesized protocols are not as efficient as the latter, the big advantage will be that they can be designed quickly (and, we hope, automatically), and verified more easily, since the synthesis algorithm has to be verified only once, and then, whenever a protocol is synthesized, it will be correct by design, as long as we verify the atomic transaction protocol separately.

4 Concluding Remarks

This paper reports work in progress in our group at the University of Utah on distributed shared memory protocol verification. We report our specific efforts relating to a recently proposed challenge problem by Lamport called the Wildfire challenge [26]. We are very impressed that the bug seeded in this protocol was recently discovered by one individual simply through inspection! No other success, either using inspection or verification tools, has, hitherto been reported. Even if one were to have luck verifying this particular protocol using today's tools, the state of the art of designing and verifying such protocols is nowhere close to being routine. Our long-term goals are to understand how such protocols are created, and to base design on *refinement*, as in our group's past work reported in [30]. Ultimately, we feel, the complexity of verification must be addressed *at design time.*

References

[1] The ASCI White Computer http://www.llnl.gov/asci/.

[2] The IBM Power4 Microarchitecture http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html.

[3] The Sun MAJC Microarchitecture http://www.sun.com/microelectronics/MAJC/.

[4] The 3GIO Third-Generation Bus Specification http://www.pcisig.com/news_room/3gio.

[5] 2001. MPV: Workshop on Specification and Verification of Shared Memory Systems, Austin, Texas, October 31, 2001 (workshop held prior to FMCAD'2000).

[6] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[7] Dennis Abts, David J. Lilja, and Steve Scott. Toward complexity-effective verification: A case study of the Cray SV2 cache coherence protocol, 2000.

[8] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[9] Ásgeir Þór Eiríksson. The formal design of 1m-gate ASICs. *Formal Methods in Systems Design*, 16(1), January 2000.

[10] L.A. Barroso, K. Gharachoroloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[11] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report COMP TR89-91, Dept. of Computer Science, Rice University, April 1989.

[12] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM TOPLAS*, 5(2):223–235, April 1983.

[13] J. B. Carter, A. Davis, R. Kuramkote, C-C. Kuo, L. B. Stoller, and M. Swanson. Avalanche: A communication and memory architecture for scalable parallel computing. In *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[14] Anne Condon and Alan J. Hu. Automatable verification of sequential consistency. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, July 2001.

[15] D. L. Dill, A, J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. pages 522–525, 1992.

[16] David L. Dill, Seungjoon Park, and Andreas Nowatzyk. Formal specification of abstract memory models. In Gaetano Borriello and Carl Ebeling, editors, *Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[17] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag, New York, NY, USA, 1996.

[18] Torbjörn Grahm and Barry Clark. Soc integration of reusable basband bluetooth ip. In *Proceedings of the 2001 Design Automation Conference*, pages 256–261, 2001.

[19] Hakan Grahn. *Evaluation of Design Alternatives for a Directory-Based Cache Coherence Protocol in Shared-Memory Multiprocessors.* PhD thesis, Lund University, October 1995.

[20] S. Gupta. Stanford dash multiprocessor: the hardware and software. In *Proc. of Parallel Architectures and Languages Europe (PARLE'92)*, pages 802–805, June 1992.

[21] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, 1978.

[22] Alan John Hu. Techniques for efficient formal verification using binary decision diagrams. Technical Report CS-TR-95-1561, 1995.

[23] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[24] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The stanford flash multiprocessor. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA '94)*, pages 302–313, April 1994.

[25] L. Lamport. Specifying concurrent systems with tla, 1999.

[26] Leslie Lamport. The wildfire challenge problem. http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html.

[27] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[28] Kenneth L. McMillan. *Symbolic Model Checking.* Kluwer Academic Press, 1993.

[29] Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem, and Ganesh Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 464–476, Vancouver, BC, Canada, June/July 1998. Springer-Verlag.

[30] Ratan Nalumasu and Ganesh Gopalakrishnan. Deriving efficient cache coherence protocols through refinement. *Formal Methods in System Design*, 1998. To appear in a Special Issue on Unity. Dominique Mery and Beverley Sanders (Editors).

[31] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, D. Lee, and M. Parkin. The s3.mp scalable shared memory multiprocessor. In *Proc. of the 27th Hawaii Int'l Conf. on System Sciences (HICSS-27)*, volume I, pages 144–153, January 1994.

[32] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1992.

[33] Sam Owre, Natarajan Shankar, and John Rushby. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction (CADE), Saratoga Springs, NY*, pages 748–752, June 1992.

[34] Seungjoon Park. *Computer Assisted Analysis of Multiprocessor Memory Systems.* PhD thesis, Stanford University, jun 1996. Department of Computer Science.

[35] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), June 1981.

[36] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. Technical report, SRC, December 2001. Research Report 176.

[37] Hemanthkumar Sivaraj. MPI port conducted in October 2001. Personal Communication.

[38] Brian R. Smith. Breaking the I/O Bottleneck.
http://www.performancecomputing.com/features/0000side.shtml.

[39] David L. Weaver and Tom Germond. *The SPARC Architecture Manual – Version 9.* P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.

Specifying and verifying fault-tolerant hardware

Scott Hazelhurst* Jean Arlat†

February 21, 2002

Abstract

Fault tolerant systems are an important class of system, often used in safety critical or highly available applications. For these systems, as well as verifying the functional and timing properties we must verify that the fault-tolerant mechanisms do protect the system in the ways expected.

This report proposes an integrated framework for specifying and verifying fault-tolerant systems: functional, timing and fault tolerance properties.. The specification is given using a temporal logic, TL, as a set of assertions of which describe the behaviour as well as the faults which should be tolerated. The faults themselves are represented as trigger-action pairs: the trigger says when a fault manifests itself, and the action says how the fault manifests itself.

The system being verified is represented as a finite state machine (FSM). The fault descriptions are used to construct observer and saboteur FSMs, which when composed with the original FSM allow a wide range of faults be modelled and fault tolerance properties verified. The verification is done using a model checking algorithm called symbolic trajectory evaluation. This framework has been implemented in the VossProver verification system, and a case study has been carried out, with promising experimental results.

1 Introduction

The importance of building fault-tolerant systems for safety-critical applications has been recognised for well over 30 years. Given their purpose, it is especially important to validate that they do have their desired or claimed fault tolerance properties. Some very successful evaluation schemes have been proposed, typically using schemes of fault-injection coupled with testing (see [11] for a discussion). Although testing-based techniques are successful, there are some limitations to these approaches: fault tolerance properties are often expressed informally; and just as exhaustively testing functional properties of a system is an intractable problem, so is testing fault tolerance properties. Though a high degree of confidence can be obtained using the appropriate testing methods, this is highly computationally intensive, and there must still be uncertainty about the result.

In other domains, formal methods have been proposed as a solution to these problems, and especially with hardware verification a large degree of success has been obtained [13]. Although formal methods have also been used in verifying fault-tolerant designs or specifications (e.g. [3, 14, 15, 18]) formal methods have not had wide-spread use in verifying fault-tolerant systems, especially verifying designs at a relatively low-level of abstraction.

This report explores the use of formal methods in specifying and verifying fault-tolerant hardware systems. The key questions explored are:

- What is a suitable language for specifying the desired fault tolerance properties?
- How can formal verification techniques be used for verifying these properties?

*School of Computer Science, University of the Witwatersrand, Johannesburg, South Africa, scott@cs.wits.ac.za

†1Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique, Toulouse, France, arlat@laas.fr

This report proposes a method of expressing fault tolerance (FT) properties of interest, and a complementary method of verifying these properties. Using these methods, the following design methodology is proposed. (1) The system is formally specified — both nominal and FT behaviour. (2) The basic non-fault-tolerant design is verified (where this design can be clearly distinguished from the fault-tolerant one). (3) The third phase is the verification of the fault-tolerant design without the presence of faults (to show that the introduction of fault tolerance has not introduced errors). (4) The final phase is the verification of the circuit in presence of faults to show that the fault tolerance mechanisms work. The main focus of this paper is the specification and verification of FT properties.

Outline: Section 2 presents the basic framework for specification and verification (based on temporal logic and model-checking). Section 3 presents a method of specifying and verifying fault tolerance properties. Section 4 presents a case study to evaluate the approach and identify its strengths and weaknesses. Section 5 concludes and suggests appropriate future research.

2 Framework for specification and verification

The choice of a specification language is difficult because there are many competing requirements of both a technical and human origin. Natural language and first-order logic are both expressive, but neither are ideal. Natural language specifications are imprecise, and first-order logic specifications may quickly become too detailed to be understandable.

This paper explores the use of a temporal logic for expressing FT properties. The major motivation for this is that temporal logics have proved very useful for specifying functional and timing properties of systems, so if a temporal logic can be used for expressing FT properties as well, a uniform framework can be provided for specification. So it is useful to know the strengths and weaknesses of a temporal logic based approach to specifying fault-tolerant systems.

2.1 The logic TL

Only a brief introduction to the logic is given here – for details see [9]. Systems are modelled as finite state machines, i.e., by a set of states $\mathcal{S}$ and by a deterministic next state function $\mathbf{Y} : \mathcal{S} \rightarrow \mathcal{S}$. (Note that the state space is modelled as a lattice which allows a certain amount of non-determinism to be expressed implicitly [6, 17].)

The specification is done using the temporal logic TL [10]. The core of TL is a set of predicates which allows the description of the instantaneous state of the system, e.g. whether a node in a circuit has a certain value or whether two nodes are related in a certain way. We denote the set of core *simple* predicates G. Typical predicates might be: $[Clk] = \mathsf{H}$ (is the clock high?); $[Reset] = \mathsf{L}$ (is the reset line low?); and $[Score] < 16$ (is the value of group of lines identified by *Score*, when considered as a bit-vector, less than 16?).

Predicates are combined using logical operators such as conjunction and negation, and temporal operators which allow us to refer to time-dependent behaviour. TL has two temporal operators: *next-time* and *until*. In practice, only the next-time operator is used. Although TL is comparatively inexpressive, it has been used successfully in a range of examples [9], and its simplicity supports a very efficient model-checking algorithm. The syntax of the logic is given by the following BNF —

$$\mathrm{TL} ::= G \mid \mathrm{TL} \wedge \mathrm{TL} \mid \neg \mathrm{TL} \mid \texttt{Next}\,\mathrm{TL} \mid \mathrm{TL}\,\texttt{Until}\,\mathrm{TL}.$$

While the truth of a predicate is evaluated with respect to a state, the truth of a TL formula is given with respect to a sequence of states, since it can refer to a number of time instants. The informal semantics is that the first state in the sequence refers to time 0 and successive states in the sequence refer to successive instants in time. The formal semantics of a formula is given by the satisfaction relation *Sat* ($Sat : \mathcal{S}^\omega \times \mathrm{TL} \rightarrow \mathcal{Q}$). Given a sequence σ and a TL formula g, *Sat* returns the truth of g with respect to the sequence σ. *Notation*: Let $\sigma = s_0 s_1 s_2 \ldots$ be a sequence in $\mathcal{S}$: then $\sigma_i = s_i$, and $\sigma_{\geq i} = s_i s_{i+1} \ldots$.

Definition 2.1. *Semantics of* TL

1. *If* $g \in G$ *then* $\mathrm{Sat}(\sigma, g) = g(s_0)$.
2. $\mathrm{Sat}(\sigma, g \wedge h) = \mathrm{Sat}(\sigma, g) \wedge \mathrm{Sat}(\sigma, h)$
3. $\mathrm{Sat}(\sigma, \neg g) = \neg \mathrm{Sat}(\sigma, g)$
4. $\mathrm{Sat}(\sigma, \mathtt{Next}\, g) = \mathrm{Sat}(\sigma_{\geq 1}, g)$
5. $\mathrm{Sat}(\sigma, g\, \mathtt{Until}\, h) = \vee_{i=0}^{\infty}(\mathrm{Sat}(\sigma_{\geq 0}, g) \wedge \ldots \wedge \mathrm{Sat}(\sigma_{\geq i-1}, g) \wedge \mathrm{Sat}(\sigma_{\geq i} h,))$

Examples and Derived Operators: Disjunction and implication are examples of derived operators. Other derived operators are possible. The most important derived operator is the `During` [] operator defined as: $\mathtt{During}\,[(f_0, t_0), \ldots, (f_n, t_n)]\ g \stackrel{\mathrm{def}}{=} \wedge_{j=0}^{n}(\wedge_{k=f_j}^{t_j} \mathtt{Next}^k g)$, which asks whether g is true from time f_0 through t_0, f_1 through $t_1, \ldots$, and from f_n through t_n. Here are some examples.

- $\mathit{diff}([\mathit{Output}], x_1 + x_2) < 2 * \mathit{delta}$. Is the absolute difference between the value on the set of lines denoted by *Output* and the sum of x_1 and x_2 less than $2 \times \mathit{delta}$? (Names of state components are in square brackets, so here *Output* is the name of a state component, while x_1, x_2 and *delta* are variables and *diff* is a function.)
- $[\mathit{Clk}] = \mathsf{H} \wedge \mathtt{Next}^{10}([\mathit{Clk}] = \mathsf{L} \wedge [\mathit{Reset}] = \mathsf{L})$: at time 0, is the clock high, and at time 10 are the clock and reset lines low?
- $\mathtt{During}\,[(0,9),(20,29)]\ ([\mathit{Clk} = \mathsf{L}]) \wedge \mathtt{During}\,[(10,19),(30,39)]\ ([\mathit{Clk} = \mathsf{H}]) \wedge$ $\mathtt{During}\,[(0,2)]\ ([\mathit{Reset}] = \mathsf{H}) \wedge \mathtt{During}\,[(3,39)]\ ([\mathit{Reset}] = \mathsf{L})$

  The formula asks if the clock is low for 10ns, then high for 10ns, then low for 10ns, and then high for 10ns; and whether the reset line is high for 3 ns (time 0 through 2 inclusive), and then low from time 3 to time 39.

2.2 Specification of systems

The specification of a system's nominal behaviour is given by a set of assertions. Each assertion consists of a pair of TL formulas, and is written like this: $\langle g \Longrightarrow\!\!\!\!\gg h \rangle$. If a model $\mathcal{M}$ satisfies this assertion, in every run of the system in which g is true, h is true too. g, the *antecedent*, can be thought of as supplying the 'input' or 'stimulus' to the circuit, while h, the *consequent* is the expected reaction to the stimulus. Where necessary we write $\mathcal{M} \models \langle g \Longrightarrow\!\!\!\!\gg h \rangle$ to emphasise that the assertion is about model $\mathcal{M}$.

Both functional and timing properties can be specified this way. For example, the following specification could describe the behaviour of a multiplication circuit, describing the result (including bit-widths), when the inputs must be stable and when the output will be stable:

$$\begin{aligned} \mathtt{During}\,[(0,9),(20,29)]\ ([\mathit{Clk} = \mathsf{L}]) \wedge \mathtt{During}\,[(10,19),(30,39)]\ ([\mathit{Clk} = \mathsf{H}]) \wedge \\ \mathtt{During}\,[(0,39)]\ [A] = x[7-0] \wedge [B] = y[7-0] \wedge \\ \Longrightarrow \qquad \mathtt{During}\,[(35,39)]\ [C] = (x \times y)[15-0] \end{aligned}$$

Significant technical detail has been omitted here. For example, the state space is a lattice – which is used for abstraction – and the truth domain is a four-valued logic rather than a boolean one. Partly the omission is for space reasons, but also because the methodology of verifying fault tolerance systems proposed here does not rely on the particular temporal logic used, or the model-checking algorithm used. Interested readers should consult [9, 10].

2.3 Symbolic Trajectory Evaluation and the VossProver Verification system

Symbolic trajectory evaluation (STE) is a model-checking algorithm due Bryant and Seger [17], and extended by Hazelhurst and Seger [9]. It is particularly suited for hardware verification, especially where accurate models of system behaviour, including timing are important. STE has a complementary compositional theory, and has been applied to a range of different circuits [9].

The VossProver verification system is built on top of Seger's Voss system [16]. The Voss system consists of three major components: an efficient implementation of binary decision diagrams [5]; an

event driven symbolic simulator with comprehensive delay and race analysis capabilities; and a general purpose, functional language called FL. STE's compositional theory has been implemented as a simple proof system in the VossProver [8]. Using FL as a script language, a verifier can interact with the proof system to either perform STE on a circuit or to use the compositional theory.

Circuits to be verified are represented internally as finite state machines. These FSMs are constructed automatically from gate-level or switch-level circuit descriptions and a number of standard input formats are supported. Voss also has its own format, called EXE. The FSM models that Voss builds are accurate models of the circuits, including timing.

3 Specification and verification of fault tolerance

3.1 Specifying fault tolerance properties

A fault is modelled with two components: a trigger and a corresponding action. The idea is that the circuit behaves normally, but that whenever the trigger is true, the behaviour of the circuit is modified (as little as possible) so as to make the action true as well. To specify fault tolerance properties, assertions are generalised to contain four pieces of information (the antecedent and consequent as before; a fault trigger; and a fault action) and is denoted thus $g \Longrightarrow\!\!\!\!\!\!\gg h$ where θ triggers ϕ (called f-assertions).

Informally, the intended meaning of $g \Longrightarrow\!\!\!\!\!\!\gg h$ where θ triggers ϕ is that in every run of the machine $\mathcal{M}$, whenever g is true, so is h *whether or not the fault described by* θ *triggers* ϕ *occurs*. Since the antecedent, consequent, trigger and action may refer to the same circuit components and variables (or different ones), an f-assertion can express a variety of behaviours in the face of faulty and non-faulty-behaviour.

In this framework, the trigger and the action can be *any* TL formulas. However, the exploratory study of Section 4 makes the following restrictions: only the non-temporal fragment of the logic can be used; and the action must non-ambiguously describe the fault for any affected nodes in the circuit.

The primary motivation of this restriction is not so much ease of implementation and efficiency of verification but simplicity of specification. The semantics of what is meant when temporal operators are used in both the trigger and action are tricky and requires some study. Examining the strengths and weaknesses of just using the non-temporal fragment of the logic is a meaningful and useful start, and indicates where extensions are necessary.

3.2 Modelling faults

So far we have modelled the finite state machine as if it were monolithic. In fact, for circuit models a convenient and efficient way to model the circuit is to represent the state of the system as a tuple, with each node (state-holding component) in the circuit making up one component of the tuple. Thus, if a circuit has n components, then $\mathcal{S} = \mathcal{C}^n$ where $\mathcal{C}$ is the set of values that an individual component can take. Similarly, the next-state function is decomposed into n next-state functions, one for each component. So, if $\mathbf{s} = \langle s_1, \ldots, s_n \rangle$, $\mathbf{Y}(\mathbf{s}) = \langle Y_1(\mathbf{s}), \ldots, Y_n(\mathbf{s}) \rangle$, with each Y_i being of type $\mathcal{S} \to \mathcal{C}$. For convenience we name each node by its index in the tuple description of the state space.

Let θ triggers ϕ be a fault. Let FA be the set of nodes that are described in ϕ and for each node $j \in FA$, let v_j be the value required for ϕ to be true. We modify each $\mathbf{Y}_i$ so that it reflects the faulty behaviour if it happens.

Formally, define $\hat{Y}_i(\mathbf{s}) \stackrel{\text{def}}{=} \begin{cases} Y_i(\mathbf{s}) & i \notin FA \\ combine(Y_i(\mathbf{s}), v_j) & i \in FA, \end{cases}$

where *combine* combines the correct value and the faulty value in the appropriate way. If $\theta(s)$ is false, *combine* produces the correct result; if $\theta(s)$ is true, *combine* produces the faulty value. (The actual implementation of *combine* is more complex than described here to take into account the lattice state-space: readers familiar with STE should note that *combine* is monotonic.

The global next state function that takes into account the fault is $\widehat{\mathbf{Y}(\mathbf{s})} \stackrel{\text{def}}{=} \langle \hat{Y}_1(\mathbf{s}), \ldots, \hat{Y}_n(\mathbf{s}) \rangle$. Finally, we can define our 'faulty' FSM to be $\widehat{\mathcal{M}} \stackrel{\text{def}}{=} (\mathcal{S}, \hat{\mathbf{Y}})$.

Note that if all temporal logic formulas were allowed in fault descriptions, then the definitions presented here would need to be generalised. This is a topic of further research.

Semantics of an f-assertion: Given a model $\mathcal{M}$, the formal semantics of an f-assertion $g \Longrightarrow\!\!\!\!\gg h$ where ϕ triggers θ is given by

$$\mathcal{M} \models g \Longrightarrow\!\!\!\!\gg h \text{ where } \phi \text{ triggers } \theta \stackrel{\text{def}}{=} \widehat{\mathcal{M}} \models \langle g \Longrightarrow\!\!\!\!\gg h \rangle.$$

3.3 Verifying f-assertions

The verification of f-assertions is accomplished by combining STE and the idea of saboteurs presented in [1]. The basic idea is as follows:

- Suppose we wish to show $\mathcal{M} \models g \Longrightarrow\!\!\!\!\gg h$ where ϕ triggers θ;
- Construct an observer machine $\mathcal{M}_O$ able to observe the state of $\mathcal{M}$. When $\mathcal{M}_O$ detects that ϕ is true of the current state of $\mathcal{M}$, it sets an internal flag to trigger the fault (see also [2, 6] for other work which has used the idea of observers);
- Construct a saboteur machine $\mathcal{M}_S$ that can inject the fault into $\mathcal{M}$. When $\mathcal{M}_O$ triggers the fault, the saboteur 'hijacks' the machine $\mathcal{M}$ and injects the fault described by θ.
- A new machine $\widehat{\mathcal{M}}$, which is the composition of $\mathcal{M}$, $\mathcal{M}_O$ and $\mathcal{M}_S$ is constructed.
- We verify $\widehat{\mathcal{M}} \models \langle g \Longrightarrow\!\!\!\!\gg h \rangle$.

All the constructions described above are done automatically and the only human intervention required is the provision of the circuit description and the f-assertion. The algorithms that perform these constructions and compositions have been implemented in the VossProver system.

4 A case study

This section explores the methodology presented in the previous sections through a case study. The system chosen as a case study is presented in [12] and described in detail in [4]. The overall architecture of the system is presented in Figure 1.

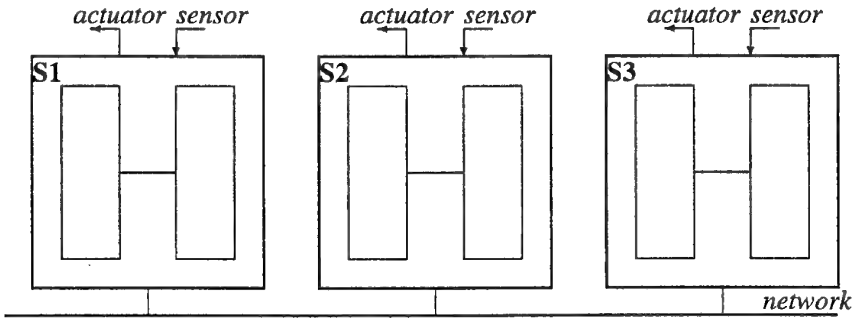


Figure 1: Overall architecture of system

The system has n *channels* (here, 3 channels, labelled *S1*, *S2* and *S3*). Each channel communicates with its environment, taking in data from sensors and then issuing commands to actuators. The channels

communicate with each other on a network. The basic premise of this system is that by implementing the system with n channels, the system is able to tolerate faults, either of sensors or of the channels themselves.

The channels operate the same protocol (described in detail in [4]). In each round of operation the channels all go through 11 phases, and one of the channels acts as master. The protocol works by each channel reading its own sensor data, broadcasting its sensor data to the other channels, followed by a process of agreeing on the data to be used and the result produced. There is also a mechanism for electing a new master.

Fault tolerance is also provided internally. Each channel consists of two *nodes* and an internal connection. One node is the *control node* that actually performs the above steps. The *monitor node* performs exactly the same steps, except that it does not communicate its results outside the channel. However, if the monitor and control produce different results, there is simple circuitry that disconnects the channel from the external network, ensuring fail-safe behaviour. Also, if either the control or monitor do not read inputs quickly enough, the channel will be extracted from the circuit. There is an error state into which a channel goes if such problems are detected; a channel only moves out of the error state if reinitialised.

The implementation examined here was based on the design described and used in [4]. That implementation was given in behavioural VHDL. The design was translated into Voss EXE format (essentially a gate-level description). Though the translation was done by hand, in principle this step could be automated. The major difference between the behavioural VHDL and EXE implementations is the way in which time is dealt with. In behavioural VHDL, it is possible to describe behaviour like 'wait $10\mu s$' directly, which is not possible at the gate-level. At the gate-level a clock has to be introduced to deal with time. For convenience of specification, only one clock is used in this implementation, but it would be straightforward (though the specification would be more cluttered) for each channel to have its own clock. For the version of the system where data and addresses are 32-bit numbers, the circuit had over 100 000 gates and 10 000 state-holding components (the implementation is rather crude!).

4.1 Verification of nominal behaviour

The specification and verification of the nominal behaviour of this system (i.e. the behaviour without faults) is an interesting exercise in its own right. However, as the main point of this paper is the specification and verification of fault-tolerant behaviour only a few points are sketched here. More detail can be found in [7].

A complete specification requires many assertions to be given. In the case study, six sample assertions were verified. The two basic goals were to show that:

- When the system is initialised, the first channel initialised is declared master, and the first round of operation is correct.
- If at the beginning of a round the system is in a consistent state, and one of the channels is the master, then the circuit works correctly and ends the round in a consistent state.

Description of clock: As this circuit is clocked, we need to refer to the clock. The TL formula, *ClockAnt* is defined to do this. The clock goes up and down for 30 clock cycles each of 100ns; for the first half of the cycle the clock is low and the second half it is high. Formally the definition is:

```
During [(0, 49), (100, 149),  ..., (2900, 2949),(3000, 3049)] Clock = F and
During [(50, 99), (150, 199), ..., (2950, 2999),(3050, 3099)] Clock = T
```

Specifying the new master: The informal specification of the circuit is that at the end of each round, that channel which had the sensor that produced the median value of all sensor values should be elected master, i.e., if the channel picked the right value, it should be the master next round. This turned out to be an interesting property to specify. A direct translation of the informal specification turns out to be wrong since more than one channel can pick the same sensor value. Instead this property is specified as:

For each channel, if it is the master then it picked the median value; and exactly one of the channels is the master.

The formal definition is given by

```
During (2700, 2749)
   (if_c S1cont:Pstatus then_c s1_got_med) & (if_c S2cont:Pstatus then_c s2_got_med) &
   (if_c S3cont:Pstatus then_c s3_got_med) & exactly_one_master_active
```

where `Sxcont:Pstatus` is a flag that indicates whether channel x is the master. The auxiliary formula `s1_got_med` is defined to be `s1[3-0]= median [s1[3-0],s2[3-0],s3[3-0]]` (and similarly for channels 2 and 3). `exactly_one_master` is just the exclusive or of the status components of each channel (the component has a high voltage if it is the master, a low voltage otherwise).

4.2 Verifying fault tolerance behaviour

As with the nominal behaviour, many aspects of the fault tolerance behaviour can be checked. A primary criterion for a specification language is that it should allow the properties to be expressed in meaningful and concise way. We need experience in specifying fault-tolerant behaviour and this is one of the goals of the paper. The examples given below illustrate the type of fault tolerance properties that can be checked.

Stuck at faults: This is a fault which always occurs, or at some time becomes true always. Here is an example of how such a fault can be modelled: **t** triggers ($[S1contnwval] = \mathbf{f}$). This says that the network validation signal of channel S1 is always false. We want to show that in this case, the other two channels still work, and agree on the right value (in this implementation, the median of two numbers is the minimum of two).

We define the antecedent *Ant* as

```
ClockAnt & network_signal=F & (During (0,10) S2init=T)&(During (11, 3099) S2init=F)&
(During (0,200)(S1init=T & S3init=T)) & (During (201,3099)  S1init=F & S3init=F) &
(During (1500, 1599) S1input=s1[2-0] & S2input=s2[2-0] & S3input=s3[2-0])
```

This is the first result proved (here we assume that the output is twice the sensor value agreed on by the channels).

```
Ant==>>
  During (2600, 2649)
    S2output=2*min[s2[2-0],s3[2-0]][3-0] and S3output=2*min[s2[2-0],s3[2-0]][3-0]
Fault assumption: stuck at fault     :  S1contnwval = F
```

Expressing relationships In the above example, the consequent is too strong in one way — another implementation might choose the maximum of two values as the median. In another sense it is too weak, since it does not really make explicit the notion of fault tolerance that we want. Here is an alternative result — logically weaker than the previous one, but it gives a more meaningful result. The essence of this result is: provided the sensor values are within a certain range of each other, then the output of the two working channels will be within some acceptable range from the median of all three sensor values.

```
Ant  ==>>
   During [(2600, 2649)] if_c well_behaved_sensors then_c well_behaved_output
Fault assumption: stuck at fault     : S1contnwval = F
```

where `well_behaved_sensors` is defined to be:

```
diff [max [s1[2-0],s2[2-0],s3[2-0]],min [s1[2-0],s2[2-0],s3[2-0]]] < delta[2-0]
```

and `well_behaved_output` is defined as

```
diff [S2output,(2*median [s1[2-0],s2[2-0],s3[2-0]])[3-0]] < 2*delta[2-0]
```

Provided the difference between the maximum and the minimum of the sensor values is δ (for any 3-bit number δ), then the output value of S2 (and analogously S3) will be within 2δ of the median of the right result if S1 fails (i.e., if S1 fails, S2 and S3 will be almost right). δ is symbolic – i.e., we verify the result for all values of δ within a certain range. Also note the care that has to be taken in specifying the bit-widths of the numbers concerned. Again there is some tedium here and it certainly clutters the specification, but it is a detail that needs to be taken care of (since the desired fault-tolerant result is not true if δ is too large, because the system does finite arithmetic). Therefore, the specification precisely describes the fault tolerance limitations.

Triggering faults on input values: The antecedent here is the same as for the stuck at fault. But here, we assert that the fault is only triggered for some values of the input. The consequent then shows that if the fault is triggered we get one result, while if it is not triggered we see the nominal behaviour ('7' is used as the synchronisation signal on the network.)

```
Ant  ==>>
    During [(2600, 2649)]
           S2output = if (s1[2-0] = 7) then (2*min [s2[2-0],s3[2-0]])[3-0]
                      else (2*median [s1[2-0],s2[2-0],s3[2-0]])[3-0]
Fault assumption --- Trigger : s1[2-0] = 7;  Fault    :  S1contnwval = F.
```

Triggering faults on state information: Faults can also be triggered on state information. This is useful when it is difficult to know when (or even if) a fault should be triggered using only input values or time. In this example, we insert an error into S1 when it gets into a state in which it is about to broadcast on the network (it broadcasts 0, rather the right sensor value). Note we force the same error into the control and the monitor node (otherwise it would automatically get extracted from the circuit).

```
Ant  ==>>  During [(2600, 2649)] S2output = (2*median [0,s2[2-0],s3[2-0]])[3-0]
Fault assumption:
Trigger: S1cont:Pstinterchange and S1cont:Pmyid; Fault: S1contdata=0 and S1montdata=0
```

Verifying fault-checking components: At a lower level of abstraction some of the circuitry that implements the fault tolerance can, of course, be checked for its functional behaviour directly. For example, in this circuit the monitor and control nodes check each other. We can show that the circuitry that performs this checking will detect errors. This is straight-forward.

4.3 Computational cost

To assess the practical worth of using formal verification, we need to consider the computational cost, since the computational costs are non-trivial. Of course, one must assess these costs in terms of the costs of not finding errors. And it is often possible to verify smaller versions of design at early stages so that even if the final verification takes many hours to run, during design and implementation the verification algorithm can be used effectively. Nevertheless, computational costs are critical.

Table 1 shows the cost of verifying the nominal and FT behaviour. The largest circuit verified had approximately 10000 state holding components and 150 000 gates. The verification was run on a 500 MHz Pentium II. Six nominal properties and six FT properties were verified, for four different versions of the circuit (varying the datapath bit-width from 4 to 32). For each run, the size of the circuit and the cost of the verifications is shown in the table.

The results show that the cost of verifying the circuit are well within the capacity of the VossProver tool (especially if one considers the fact the figures are inflated by the overheads of loading the system, building the circuit etc, which usually only has to be done once a session). The cost of verification appears to grow more quickly than the number of gates (but still logarithmic in the size of the state space, though a more through analysis is needed here).

Bit-width	4	8	16	32
Number of gates	26000	50000	85000	150000
Cost for nominal properties (s)	63	85	128	308
Cost for FT properties (s)	19	31	69	275

Table 1: Cost of verification of nominal and FT properties in seconds

Human cost: Specification requires significant human insight. The verification of all results shown here is completely automatic (though this is not something that we can expect at this stage in general).

5 Conclusion

This paper has examined the use of formal methods in verifying fault-tolerant systems' designs, presenting an approach to specifying and verifying fault-tolerant systems. The logic used also allows a range of fault conditions to be expressed. These fault conditions are given as trigger-action pairs: the trigger indicates when a fault will occur and the action says what type of fault occurs. Using the ideas of saboteurs [1], the method of symbolic trajectory evaluation can be generalised to be able to verify FT properties.

A case study was performed to evaluate the proposed approach. Overall, the case study shows that the approach is successful. The nominal behaviour of the circuit was verified, and then a range of FT properties were examined including: stuck at faults; faults triggered by input values; and faults triggered by state conditions.

In addition, the expected behaviour could be modelled exactly (e.g., the output is x) or approximately (e.g., the output is within a certain range). It is also possible to verify directly that certain fault-monitoring circuitry performs its task. The experimental results also showed that the computational costs of STE were quite reasonable.

Future research: There are a number of issues for future research:

Language for specifying fault tolerance: The case study explored the use of the non-temporal fragment of TL for specifying the trigger-action pairs. This proved capable of expressing a range of different behaviour. We need to do more case studies to get experience in what type of FT properties need to be proved, in order to find where the limits are. It appears that allowing temporal operators in both triggers and actions would be useful, and the framework of using observers and saboteurs should be able to cope with the extension. One particular anticipated difficulty is the specification of both absolute and relative times in the fault specification.

Use for determining fault tolerance coverage: One interesting possibility is to generalise the specification of the fault and/or antecedent, expecting the verification to fail. The point of this is that the information given by the VossProver explaining why the verification failed could be used to determine fault tolerance coverage.

Compositional theory: A very important technique to overcome the state explosion problem that bedevils symbolic model checking is the use of compositionality. STE has a simple but successful compositional theory that has been implemented in the VossProver [8]. If we wish to use STE to deal with fault tolerance we need to extend the theory for combining assertions to a theory that deals with combining f-assertions.

Improving the performance of algorithms: And, of course, all algorithms can benefit from improvement. Even though in the case study chosen the STE algorithm could easily prove the required assertions and f-assertions, the insatiable demands for memory and CPU cycles needs to be met . . . .

Acknowledgements:

This work was funded in part by the South African National Research Foundation and the Centre National de la Recherche Scientifique.

References

[1] J. Arlat, J. Boué, and Y. Crouzet. Validation-based Development of Dependable Systems. *IEEE Micro*, 19(4):66–79, Jul-Aug 1999.

[2] I. Beer, S. Ben-David, D. Geist, R. Gewirtzman, and M. Yoeli. Methodology and system for practical formal verification of reactive hardware. In D.I. Dill, editor, *CAV '94: Proceedings of the Sixth International Conference on Computer Aided Verification*, Lecture Notes in Computer Science 818, pages 182–193, Berlin, June 1994. Springer-Verlag.

[3] C. Bernadeschi, A. Fantechi, S. Gnesi, and A. Santone. Formal validation of fault tolerance mechanisms. In *Digest of FastAbstracts of the 28th International Symposium on Fault-Tolerant Computing*, pages 66–67. IEEE Computer Society Press, 1998. http://www.chillarege.com/ftcs/fastabstracts/389.html.

[4] J. Boué. *Test de la Tolérance aux fautes par injection de fautes dans des modèles du simuation VHDL.* PhD thesis, Institut National Polytechnique de Toulouse, November 1997. LAAS Report 97503.

[5] R.E. Bryant. Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] S. Hazelhurst. *Compositional Model Checking of Partially-Ordered State Spaces.* PhD thesis, University of British Columbia, Department of Computer Science, 1996.

[7] S. Hazelhurst and J. Arlat. Specifying and verifying fault-tolerant hardware. LAAS Report 99514, Laboratoire d'Analyse et d'Architecture des Systèmes, Centre National de la Recherche Scientifique, December 1999.

[8] S. Hazelhurst and C.-J.H. Seger. A Simple Theorem Prover Based on Symbolic Trajectory Evaluation and BDD's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 14(4):413–422, April 1995.

[9] S. Hazelhurst and C.-J.H. Seger. Symbolic Trajectory Evaluation. In Kropf [13], pages 3–79.

[10] S. Hazelhurst and C.-J.H. Seger. Model checking lattices: Using and reasoning about information orders for abstraction. *Logic Journal of the IGPL*, 7(3):375–411, May 1999.

[11] M.-C. Hsueh, T. K. Tsai, and R. K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, April 1997.

[12] H. Kopetz. The time-triggered approach in real-time system design. In B. Randell, J.-C. Laprie, H. Kopetz, and B. Littlewood, editors, *Predictably Dependable Computing Systems*, Basic Research, pages 53–66. Springer, 1995.

[13] T. Kropf, editor. *Formal Hardware Verification: Methods and Systems in Comparison.* State of the Art Survey Lecture Notes in Computer Science 1287. Springer-Verlag, Berlin, 1997.

[14] S. Owre, J. Rushby, N. Shankar, and F. von Henke. Formal Verification for Fault-Tolerant Architectures: Prolegomena to the Design of PVS. *IEEE Transactions on Software Engineering*, 21(2):107–125, February 1995.

[15] J. Rushby. Systematic Formal Verification for Fault-Tolerant Time-Triggered Algorithms. *IEEE Transactions on Software Engineering*, 25(5):651–660, 1999.

[16] C.-J.H. Seger. Voss — A Formal Hardware Verification System User's Guide. Technical Report 93-45, Department of Computer Science, University of British Columbia, November 1993. Available by anonymous ftp as ftp://ftp.cs.ubc.ca/pub/local/techreports/1993/TR-93-45.ps.gz.

[17] C.-J.H. Seger and R.E. Bryant. Formal Verification by Symbolic Evaluation of Partially-Ordered Trajectories. *Formal Methods in Systems Design*, 6:147–189, March 1995.

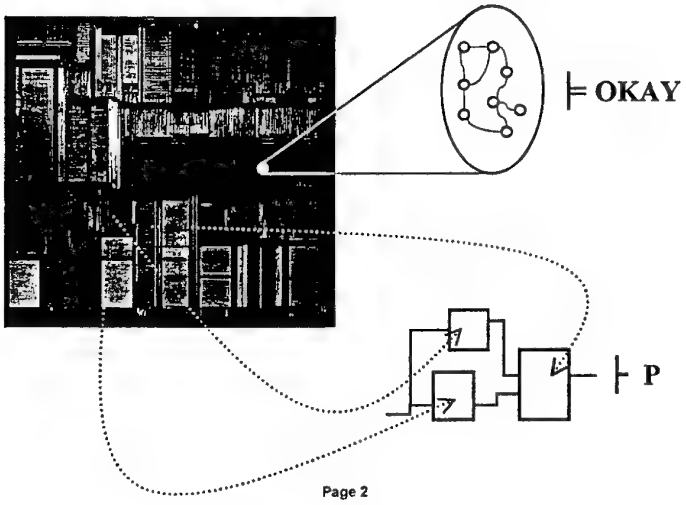
[18] M. Sheeran and G. Stålmarck. A tutorial on Stålmarck's procedure for propositional logic. *Formal Methods in System Design*, 16(1):23–58, January 2000.

A Stream-based Framework for Reasoning with STE and other LTL Verification Formalisms

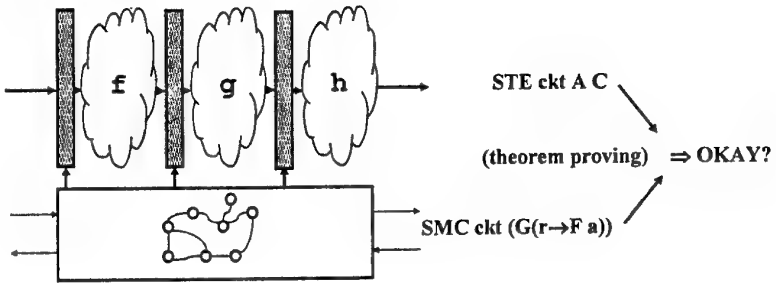
John O'Leary
Strategic CAD Labs
Intel Corporation
5200 NE Elam Young Pkwy
Hillsboro, OR 97124, USA
joleary@ichips.intel.com

Tom Melham
Dept of Computing Science
University of Glasgow
Glasgow, Scotland, G12 8QQ
tfm@dcs.gla.ac.uk

Model checking and theorem proving



Motivation



- Use the tool most suited for each sub-task
- Express and verify properties beyond the reach of a single model checking run

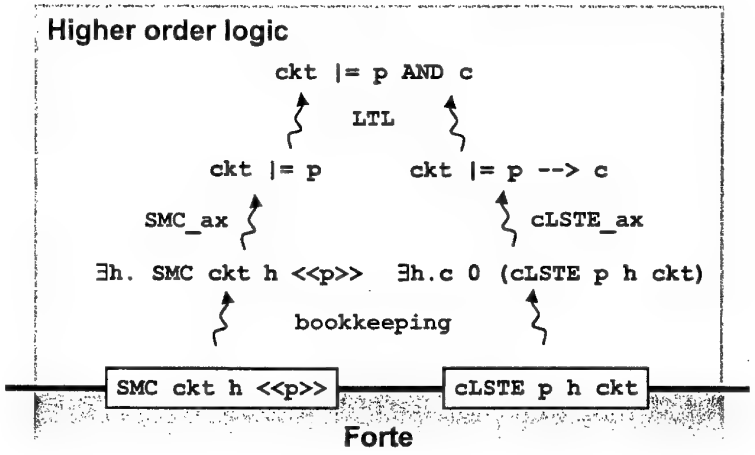
Page 3

Our approach

- Combine STE and SMC in a higher order logic theorem prover
  - A shallow embedding enables semantic reasoning about verification results
- Employ reflection to
  - Make links between model checking and higher order logic *explicit* axioms rather than implicit in the metalogic
  - Enable a lightweight, logically-coherent bookkeeping system for model-checking runs
  - Make calls to model checkers *explicit* in proofs

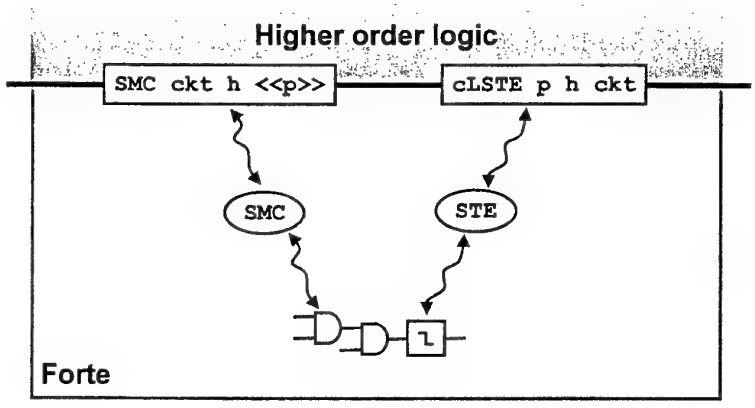
Page 4

Logical architecture



Page 5

Physical architecture



Page 6

LTL

- A subset of Intel's ForSpec Temporal Language

LTL f ::= value n $\quad n \in Nd$
| $f1 \wedge f2$
| $\neg f$.
| NEXT f
| $f1$ UNTIL $f2$
| ...

Page 7

LTL semantics

- State (assigning each node a Boolean value)

  St = Nd → bool

- Trace (a sequence of states)

  Tr = nat → (Nd → bool)

- Satisfaction

  _ sat _ :: (nat, Tr) → LTL → bool

- Examples:

  i,σ sat [value n] iff σ i n
  i,σ sat [$f1 \wedge f2$] iff i,σ sat [$f1$] and i,σ sat [$f2$]
  i,σ sat [NEXT f] iff i+1,σ sat [f]

Page 8

LTL in higher order logic

- LTL operators are shallowly embedded as functions of type nat → Tr → bool
- Example:

  i,σ sat [*f1* ∧ *f2*] iff i,σ sat [*f1*] and i,σ sat [*f2*]

```
p && q =def
    \t.\s. (p t s) AND (q t s);
```

Page 9

LTL in higher order logic

- A circuit satisfies a property iff all its traces do
- We can treat `in_L :: Tr->fsm->bool` as an uninterpreted predicate for our purpose

```
ckt |= p =def =
   Forall s. (s in_L ckt) ==> p 0 s;
```

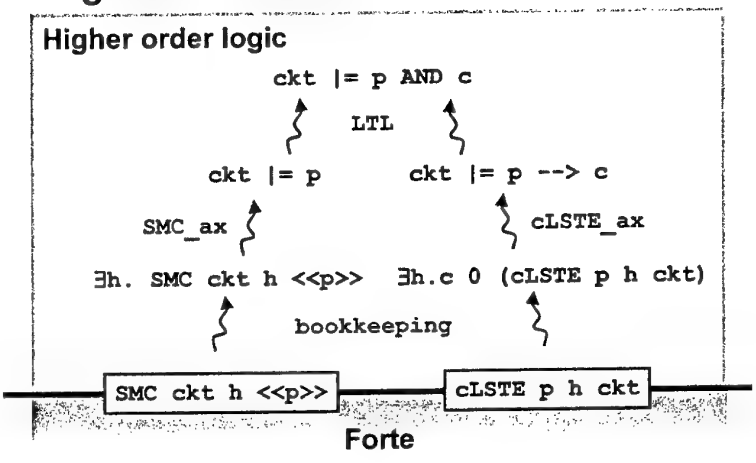
Page 10

Reasoning in LTL

```
NEXT_and =
    |- Forall p. Forall q.
          NEXT (p && q)
          =
          (NEXT p) && (NEXT q)

time_shift =
  |- Forall ckt. Forall p.
          (ckt |= p)
          ==>
          (ckt |= ALWAYS p)
```

Page 11

Logical architecture



Page 12

SMC

`SMC ckt usefuls frees <<p>>`

- constructs a specification automaton from the LTL property p using standard tableau construction, runs the model checker and returns T/F
- usefuls and frees are hints for pruning the circuit
- Reflection: <<p>> means "the syntax of p"

Page 13

Interface to SMC, using reflection

```
SMC_ax def=
  |- Forall ckt. Forall p.
       (WF <<p>>) AND
       (Exists usefuls. Exists frees.
           SMC ckt usefuls frees <<p>>)
       ==>
       (ckt |= p)
```

Page 14

Conventional solution

- Interface to SMC is implicit:
  - A metafunction `mk_SMC_ax` takes the circuit, pruning info, and property as arguments, calls SMC, and generates the appropriate axiom if SMC succeeds

```
mk_SMC_ax ckt [] [] <<spec>>
```

```
|- (ckt |= spec)
```

Page 15

STE

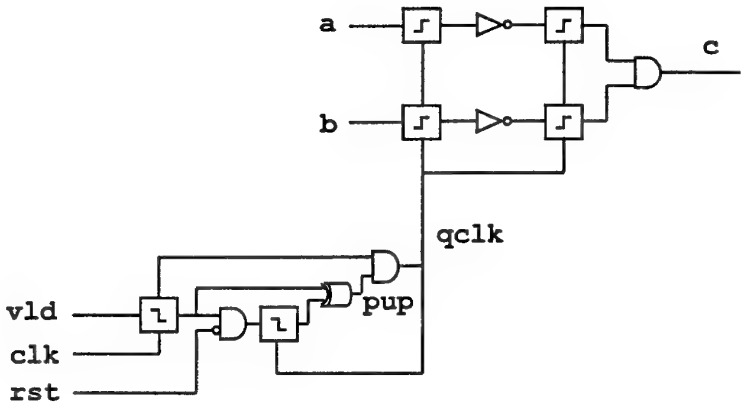
`cLSTE assump hints ckt`

- provides a trace-based interface to standard STE
- `cLSTE` returns a trace `(int->string->bool)` generated under assumption `asm` that can be used to check satisfaction of a bounded LTL formula

```
cLSTE_ax ≝
  |- Forall ckt. Forall asm. Forall p.
       (Exists hints.
            p 0 (cLSTE asm hints ckt))
       ==>
       (ckt |= (asm --> p))
```

Page 16

Example circuit



Page 17

Helpful definitions

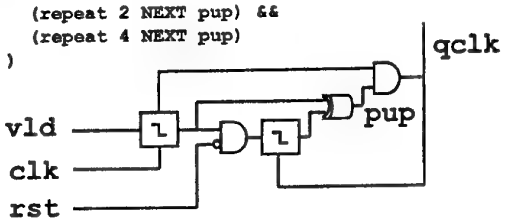
- It is convenient to introduce these definitions

```
let falling c = c && NEXT (! c);
let rising  c = (! c) && NEXT c;

letrec repeat 0 f p = p
 /\    repeat n f p = f (repeat (n-1) f p);

let weak_mutex a b = ! (a && b);
```

Page 18

Control property

```
let ctl_spec =
  (ALWAYS (clk ~= ! (NEXT clk))) &&
  (ALWAYS (falling clk -->
             weak_mutex vld (repeat 2 NEXT vld)))
  -->
  ALWAYS (
    (falling clk) && (value "rst") -->
      ALWAYS (
        (falling clk && vld &&
         ! (value "rst") && repeat 2 NEXT (! (value "rst")))
        -->
        (repeat 2 NEXT pup) &&
        (repeat 4 NEXT pup)
      )
  );
```



Page 19

Proving the control property

Evaluation

```
|- WF <<ctl_spec>> AND
   SMC ckt ["c2c"] [] <<ctl_spec>>
```

"bookkeeping"

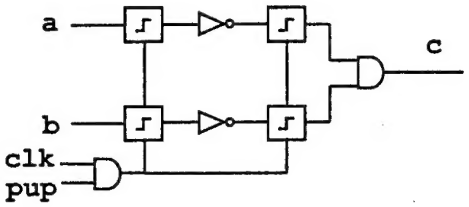
```
|- WF <<ctl_spec>> AND
   (Exists usefuls. Exists frees.
      SMC ckt usefuls frees <<ctl_spec>>)
```

SMC_ax

```
|- ckt |= ctl_spec
```

Page 20

Datapath property




```
let dp_spec =
  (ALWAYS (clk ~= ! (NEXT clk)))  // dp_spec_a
  -->
  ALWAYS (                        // dp_spec_c
    (falling clk &&
     repeat 2 NEXT pup && repeat 4 NEXT pup)
    -->
    ((repeat 5 NEXT c) ~= ! ((NEXT a) || (NEXT b))));
```

Page 21

Proving the datapath property

Evaluation

```
|- dp_spec_c 0
       (cLSTE dp_spec_a dp_hints ckt)
```

"bookkeeping"

```
|- Exists hints.
     dp_spec_c 0
       (cLSTE dp_spec_a hints ckt)
```

cLSTE_ax

```
|- ckt |= dp_spec_a -> dp_spec_c
```

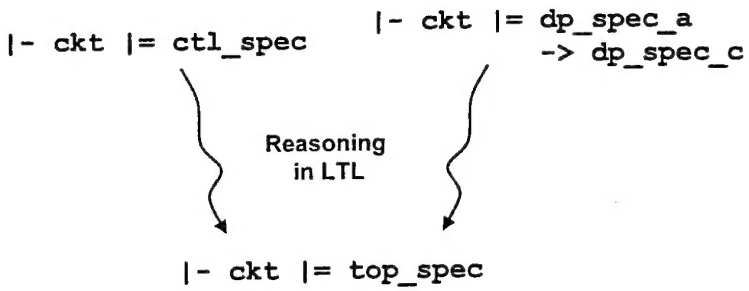
Page 22

Combined property

```
let top_spec =
  (ALWAYS (clk ~= ! (NEXT clk))) &&
  (ALWAYS (falling clk -->
             weak_mutex vld (repeat 2 NEXT vld)))
  -->
  ALWAYS (
    (falling clk) && (value "rst")
    -->
    ALWAYS (
      (falling clk && vld &&
       !(value "rst") && repeat 2 NEXT (!(value "rst")))
      -->
      ((repeat 5 NEXT c) ~= ! ((NEXT a) || (NEXT b)))));
```

Page 23

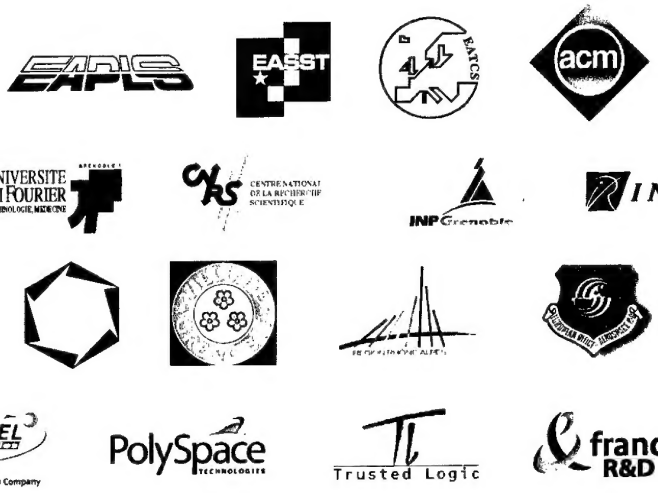
Proving the combined property



Page 24

Summary

- STE, SMC are combined within a higher order logic theorem prover with reflection
  - Calls to model checkers are explicit in proofs
    - Including directives, hints, etc
  - Theorem prover is used as a bookkeeping tool to manage model checking runs
    - Abstracting away directives, hints
    - Extensions can handle simple reasoning: case-splitting, assume-guarantee
  - Explicit links between STE, SMC and specification logic
  - Spec logic is shallowly embedded in higher order logic
    - Full power of the theorem prover is available for complex reasoning



Location: Domaine Universitaire, GRENOBLE